(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 704 796 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
03.04.1996 Bulletin 1996/14

(51) Int Cl.6: G06F 9/46

(21) Application number: 95304188.6

(22) Date of filing: 16.06.1995

(84) Designated Contracting States:
AT BE CH DE ES FR GB IT LI NL SE

(30) Priority: 28.09.1994 US 263313

(71) Applicant: International Business Machines Corporation
Armonk, N.Y. 10504 (US)

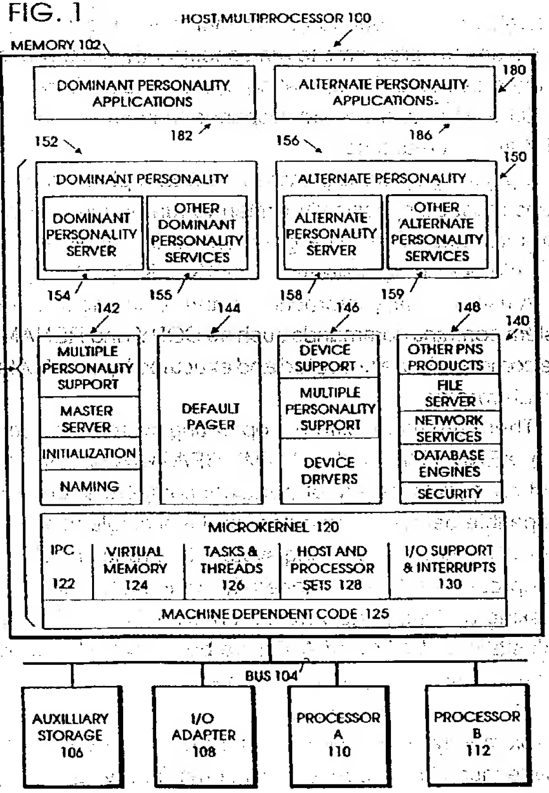
(72) Inventors:
- Magee, James Michael
Lake Worth, Florida 33463 (US)
- Sotomayor, Guy Gil, Jr.
West Palm Beach, Florida 33415 (US)
- Youngworth, Christopher Dean
Savoy, Illinois 61874 (US)

(74) Representative: Williams, Julian David
IBM United Kingdom Limited,
Intellectual Property Department,
Hursley Park
Winchester, Hampshire SO21 2JN (GB)

(54) Capability engine method and apparatus for a microkernel data processing system

(57) A microkernel interprocess communication subsystem and method provide fast and efficient communication between clients and servers in uniprocessing, multiprocessing, and distributed processing environments. A microkernel operating system includes a capability engine module that manages capabilities or rights to map regions of the memory shared by multiprocessing tasks. There is a wide range of port rights that can be attributed to a task port; various permission levels, security levels, priority levels, processor and resource availability, etc. The capability engine analyses these rights and selectively enables transfers between tasks. In this manner, the capability engine manages the interprocess communication that must take place between the many clients and servers in a Microkernel System, in a fast and efficient manner.

FIG. 1



Description

The present invention broadly relates to data processing systems and more particularly relates to improvements in operating systems for data processing systems.

The invention disclosed herein is related to the copending United States Patent Application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, which is entitled "METHOD AND APPARATUS FOR MANAGEMENT OF MAPPED AND UNMAPPED REGIONS OF MEMORY IN A MICROKERNEL DATA PROCESSING SYSTEM", Serial Number 263,710, filed on the same day as the instant application, IBM Docket Number BC9-94-053, assigned to the International Business Machines Corporation, and incorporated herein by reference.

The operating system is the most important software running on a computer. Every general purpose computer must have an operating system to run other programs. Operating systems typically perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disc, and controlling peripheral devices such as disc drives and printers. For more complex systems, the operating system has even greater responsibilities and powers. It makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also typically responsible for security, ensuring that unauthorized users do not access the system.

Operating systems can be classified as multi-user operating systems, multi-processor operating systems, multi-tasking operating systems, and real-time operating systems. A multi-user operating system allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users. A multi-processing program allows a single user to run two or more programs at the same time. Each program being executed is called a process. Most multi-processing systems support more than one user. A multi-tasking system allows a single process to run more than one task. In common terminology, the terms multi-tasking and multi-processing are often used interchangeably even though they have slightly different meanings. Multi-tasking is the ability to execute more than one task at the same time, a task being a program. In multi-tasking, only one central processing unit is involved, but it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same time. There are two basic types of multi-tasking, preemptive and cooperative. In preemptive multi-tasking, the operating system parcels out CPU time slices to each program. In cooperative multi-tasking, each program can control the CPU for as long as it needs it. If a program is not using the CPU however, it can allow another program to use it temporarily. For example, the OS/2 (TM) and UNIX (TM) operating systems use preemptive multi-tasking, whereas the Multi-Finder (TM) operating system for Macintosh (TM) computers uses cooperative multi-tasking. Multi-processing refers to a computer system's ability to support more than one process or program at the same time. Multi-processing operating systems enable several programs to run concurrently. Multi-processing systems are much more complicated than single-process systems because the operating system must allocate resources to competing processes in a reasonable manner. A real-time operating system responds to input instantaneously. General purpose operating systems such as DOS and UNIX are not real-time.

Operating systems provide a software platform on top of which application programs can run. The application programs must be specifically written to run on top of a particular operating system. The choice of the operating system therefore determines to a great extent the applications which can be run. For IBM compatible personal computers, example operating systems are DOS, OS/2 (TM), AIX (TM), and XENIX (TM).

A user normally interacts with the operating system through a set of commands. For example, the DOS operating system contains commands such as COPY and RENAME for copying files and changing the names of files, respectively. The commands are accepted and executed by a part of the operating system called the command processor or command line interpreter.

There are many different operating systems for personal computers such as CP/M (TM), DOS, OS/2 (TM), UNIX (TM), XENIX (TM), and AIX (TM). CP/M was one of the first operating systems for small computers. CP/M was initially used on a wide variety of personal computers, but it was eventually overshadowed by DOS. DOS runs on all IBM compatible personal computers and is a single user, single tasking operating system. OS/2, a successor to DOS, is a relatively powerful operating system that runs on IBM compatible personal computers that use the Intel 80286 or later microprocessor. OS/2 is generally compatible with DOS but contains many additional features, for example it is multi-tasking and supports virtual memory. UNIX and UNIX-based AIX run on a wide variety of personal computers and work stations. UNIX and AIX have become standard operating systems for work stations and are powerful multi-user, multi-processing operating systems.

In 1981 when the IBM personal computer was introduced in the United States, the DOS operating system occupied approximately 10 kilobytes of storage. Since that time, personal computers have become much more complex and require much larger operating systems. Today, for example, the OS/2 operating system for the IBM personal computers can occupy as much as 22 megabytes of storage. Personal computers become ever more complex and powerful as time goes by and it is apparent that the operating systems cannot continually increase in size and complexity without imposing a significant storage penalty on the storage devices associated with those systems.

It was because of this untenable growth rate in operating system size, that the MACH project was conducted at the Carnegie Mellon University in the 1980's. The goal of that research was to develop a new operating system that would allow computer programmers to exploit modern hardware architectures emerging and yet reduce the size and the number of features in the kernel operating system. The kernel is the part of an operating system that performs basic functions such as allocating hardware resources. In the case of the MACH kernel, five programming abstractions were established as the basic building blocks for the system. They were chosen as the minimum necessary to produce a useful system on top of which the typical complex operations could be built externally to the kernel. The Carnegie Mellon MACH kernel was reduced in size in its release 3.0, and is a fully functional operating system called the MACH microkernel. The MACH microkernel has the following primitives: the task, the thread, the port, the message, and the memory object. The task is the traditional UNIX process which is divided into two separate components in the MACH microkernel. The first component is the task, which contains all of the resources for a group of cooperating entities. Examples of resources in a task are virtual memory and communications ports. A task is a passive collection of resources; it does not run on a processor.

The thread is the second component of the UNIX process, and is the active execution environment. Each task may support one or more concurrently executing computations called threads. For example, a multi-threaded program may use one thread to compute scientific calculations while another thread monitors the user interface. A MACH task may have many threads of execution, all running simultaneously. Much of the power of the MACH programming model comes from the fact that all threads in a task share the task's resources. For instance, they all have the same virtual memory (VM) address space. However, each thread in a task has its own private execution state. This state consists of a set of registers, such as general purpose registers, a stack pointer, a program counter, and a frame pointer.

A port is the communications channel through which threads communicate with each other. A port is a resource and is owned by a task. A thread gains access to a port by virtue of belonging to a task. Cooperating programs may allow threads from one task to gain access to ports in another task. An important feature is that they are location transparent. This capability facilitates the distribution of services over a network without program modification.

The message is used to enable threads in different tasks to communicate with each other. A message contains collections of data which are given classes or types. This data can range from program specific data such as numbers or strings to MACH related data such as transferring capabilities of a port from one task to another.

A memory object is an abstraction which supports the capability to perform traditional operating system functions in user level programs, a key feature of the MACH microkernel. For example, the MACH microkernel supports virtual memory paging policy in a user level program. Memory objects are an abstraction to support this capability.

All of these concepts are fundamental to the MACH microkernel programming model and are used in the kernel itself. These concepts and other features of the Carnegie Mellon University MACH microkernel are described in the book by Joseph Boykin, et al, "Programming Under MACH", Addison Wessely Publishing Company, Incorporated, 1993.

Additional discussions of the use of a microkernel to support a UNIX personality can be found in the article by Mike Accetta, et al, "MACH: A New Kernel Foundation for UNIX Development", Proceedings of the Summer 1986 USENIX Conference, Atlanta, Georgia. Another technical article on the topic is by David Golub, et al, "UNIX as an Application Program", Proceedings of the Summer 1990 USENIX Conference, Anaheim, California.

The above cited, copending patent application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, describes the Microkernel System 115 shown in Figure 1, which is a new foundation for operating systems. The Microkernel System 115 provides a concise set of kernel services implemented as a pure kernel and an extensive set of services for building operating system personalities implemented as a set of user-level servers. The Microkernel System 115 is made up of many server components that provide the various traditional operating system functions and that are manifested as operating system personalities. The Microkernel System 115 uses a client/server system structure in which tasks (clients) access services by making requests of other tasks (servers) through messages sent over a communication channel. Since the microkernel 120 provides very few services of its own (for example, it provides no file service), a microkernel 120 task must communicate with many other tasks that provide the required services. This raises the problem of how to manage the interprocess communication that must take place between the many clients and servers in the system, in a fast and efficient manner.

In accordance with the present invention, there is now provided a method for interprocess communication in a microkernel architecture data processing system, comprising the steps of: loading a microkernel including a capability engine module into a memory of the data processing system; forming with said microkernel a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object; forming with said microkernel a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights; registering in said capability engine, said first set of port rights for said first task container and said second set of port rights for said second task container; comparing said first set of port rights and said second set of port rights in said capability engine; and enabling a transfer with said capability engine, of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object.

Viewing the present invention from a second aspect, there is provided a subsystem for interprocess communication in a microkernel architecture data processing system, comprising: a microkernel in a memory of a data processing system, including a capability engine module; a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object; a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights; said capability engine registering said first set of port rights for said first task container and said second set of port rights for said second task container; said capability engine comparing said first set of port rights and said second set of port rights; and said capability enabling a transfer of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object.

Viewing the present invention from a third aspect, there is provided A system for interprocess communication in a microkemel architecture, comprising a memory means in a data processing system, for storing data and programmed instructions; an interprocess communications means in said memory means, for coordinating message passing between tasks in said memory means; a first task in said memory means having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task to access said memory object; a second task in said memory means having a set of attributes defining a second communication port and a second set of port rights; a processor means coupled to said memory means, for executing said programmed instructions; a first thread in said memory means associated with said first task, for providing said programmed instructions for execution in said processor means; a capability engine means in said interprocess communications means, for registering said first set of port rights for said first task and said second set of port rights for said second task; said thread providing a message from said first task to said interprocess communications means, for providing said pointer to said second task; said capability engine means comparing said first set of port rights and said second set of port rights; and said capability engine means enabling a transfer of said pointer and said first port rights from said first task to said second task to confer onto said second task said capability to access said memory object.

Viewing the present invention from a fourth aspect, there is provided a system for interprocess communication in a microkemel architecture, comprising: a memory means in a data processing system, for storing data and programmed instructions; a microkernel means in said memory means, for coordinating operations between a plurality of tasks in said memory means; an interprocess communications means in said microkernel means, for coordinating message passing between tasks in said memory means; a first task in said memory means having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task to access said memory object; a second task in said memory means having a set of attributes defining a second communication port and a second set of port rights; a processor means coupled to said memory means, for executing said programmed instructions; a first thread in said memory means associated with said first task, for providing said programmed instructions for execution in said processor means; a capability engine means in said interprocess communications means, for registering said first set of port rights for said first task and said second set of port rights for said second task; said thread providing a message from said first task to said interprocess communications means, for providing said pointer to said second task; said capability engine means comparing said first set of port rights and said second set of port rights; and said capability engine means enabling a transfer of said pointer and said first port rights from said first task to said second task to confer onto said second task said capability to access said memory object.

Viewing the present invention from a fifth aspect, there is provided a system for interprocessor communication in a shared memory multiprocessor, comprising: a memory means in a data processing system, for storing data and programmed instructions; a microkernel means in said memory means, for coordinating operations between a plurality of tasks in said memory means; an interprocess communications means in said microkernel means, for coordinating message passing between tasks in said memory means; a first task in said memory means having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task to access said memory object; a first processor means coupled to said memory means, for executing said programmed instructions; a first thread in said memory means associated with said first task, for providing said programmed instructions for execution in said first processor means; a second task in said memory means having a set of attributes defining a second communication port and a second set of port rights; a second processor means coupled to said memory means, for executing said programmed instructions; a second thread in said memory means associated with said second task, for providing said programmed instructions for execution in said second processor means; a capability engine means in said interprocess communications means, for registering said first set of port rights for said first task and said second set of port rights for said second task; said first thread providing a message from said first task to said interprocess communications means, for providing said pointer to said second task; said capability engine means comparing said first set of port rights and said second set of port rights; and said capability engine means enabling a transfer of said pointer and said first port rights from said first task to said second task to confer onto said second task said capability to access said memory object.

Viewing the present invention from a sixth aspect, there is provided a system for interprocessor communication in a distributed processor system, comprising: a memory means in a first host system of a distributed processor system, for storing data and programmed instructions; a microkernel means in said memory means, for coordinating operations between a plurality of tasks in said memory means; an interprocess communications means in said microkernel means, for coordinating message passing between tasks in said memory means; a first task in said memory means having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task to access said memory object; a first processor means in said first host system, coupled to said memory means, for executing said programmed instructions; a first thread in said memory means associated with said first task, for providing said programmed instructions for execution in said first processor means; a second task in said memory means having a set of attributes defining a second communication port and a second set of port rights; a second thread in said memory means associated with said second task, for providing said programmed instructions for execution in said first processor means; a capability engine means in said interprocess communications means, for registering said first set of port rights for said first task and said second set of port rights for said second task; said first thread providing a message from said first task to said interprocess communications means, for providing said pointer to said second task; said capability engine means comparing said first set of port rights and said second set of port rights; and said capability engine means enabling a transfer of said pointer and said first port rights from said first task to said second task to confer onto said second task said capability to access said memory object; a communications link coupling said first processor in said first host system to a second host system of said distributed processor system; a second processor means in said second host system, coupled to said first processor means over said communications link; said second thread providing a reference to said pointer from said second task to said communications link, for providing said reference to said pointer to said second processor means; said second processor means receiving said reference to said pointer over said communications link from said first processor means, to enable said second processor means to access said memory object in said first host system.

The present invention this provides an improved microkernel architecture for a data processing system. Specifically, the present invention provides an improved microkernel architecture for a data processing system that is more simplified in its interprocess communication operations than has been capable in the prior art. The present invention this provides an improved microkernel architecture for a data processing system, that has a faster and more efficient interprocess communication capability. Still furthermore, the present invention to provide an improved microkernel architecture for a data processing system, that has greater flexibility in the exchange of messages between tasks within a shared memory environment and between distributed data processors that do not share a common memory.

A preferred embodiment of the present invention provides a method and subsystem are disclosed for interprocess communication in a microkernel architecture data processing system. The data processing system can be a shared memory multiprocessing system or a uniprocessor system. A microkernel operating system is loaded into the memory of the data processing system. In accordance with the invention, the microkernel includes a capability engine module that manages capabilities or rights to map regions of the memory. A first task container is formed by the microkernel in the memory, having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, the first set of port rights conferring a capability on the first task container to access the memory object. A second task container is formed in the memory having a set of attributes defining a second communication port and a second set of port rights.

In a preferred embodiment of the present invention, the capability engine registers the first set of port rights for the first task container and the second set of port rights for the second task container. The capability engine then compares the first set of port rights and the second set of port rights to determine if the second task can be allowed to gain access to the memory object. There is a wide range of port rights that can be attributed to a task port; various permission levels, security levels, priority levels, processor and other resource availability, and many others, limited only by the user's imagination. The capability engine analyses these rights and can selectively enable a transfer of the pointer and the first port rights from the first task container to the second task container to confer onto the second task container the capability to access the memory object. In this manner, the capability engine manages the interprocess communication that must take place between the many clients and servers in the Microkernel System, in a fast and efficient manner.

The present invention applies to uniprocessors, shared memory multiprocessors, and multiple computers in a distributed processor system.

Preferred embodiments of the present invention will now be described with reference to the accompanying figures, in which:

Figure 1 is a functional block diagram of the Microkernel System 115 in the memory 102 of the host multiprocessor 100, showing how the microkernel and personality-neutral services 140 run multiple operating system personalities on a variety of hardware platforms;

Figure 2 shows the client visible structure associated with a thread;

Figure 3 shows the client visible task structures;

Figure 4 shows a typical port, illustrating a series of send rights and the single receive right;

Figure 5 shows a series of port rights, contained in a port name space or in transit in a message;

Figure 6 shows the client visible virtual memory structures;

Figure 7 shows a functional block diagram of the host multiprocessor system, with the IPC subsystem and the capability engine managing interprocess communications between two tasks with threads running on two processors;

Figure 8 shows a functional block diagram of two host multiprocessor systems running in a distributed processing arrangement, with the IPC subsystem and the capability engine on each host processor managing interprocess communications between tasks with the exchange of messages between the two hosts over a communications link;

Figure 9 shows a simple message transfer with the IPC;

Figure 10 shows an example message transfer using the capability engine and the message passing library;

Figure 11 shows an example message transfer using the capability engine and the message passing library;

Figure 12 shows an example message transfer using the capability engine and the message passing library;

Figure 13 shows an example message transfer using the capability engine and the message passing library;

Figure 14 shows an example message transfer using the capability engine and the message passing library;

Figure 15 shows an example message transfer using the capability engine and the message passing library;

Figure 16 shows an example message transfer using the capability engine and the message passing library;

Figure 17 shows an example message transfer using the capability engine and the message passing library;

Figure 18 is a diagram outlining message layout;

Figure 19 illustrates a typical call of a by-proxy user level library;

Figure 20 illustrates a message control structure;

Figure 21 illustrates a message header structure;

Figure 22 illustrates an example of a trusted client/known message ID;

Figure 23 illustrates an example of a non-trusted client/known message ID;

Figure 24 illustrates a message format registration;

Figure 25 illustrates the overwrite buffer operation;

Figure 26 illustrates an RPC transfer;

Figure 27 illustrates queue support through the capability engine;

Figure 28 illustrates the basic execution loop of the multiplexing server;

Figure 29 is the message passing library anonymous reply algorithm;

Figure 30 illustrates share region initialization;

Figure 31 illustrates share region usage in the RPC common case.

Part A. The Microkernel System

Section 1. Microkernel Principles

Figure 1 is a functional block diagram of the Microkernel System 115, showing how the microkernel 120 and personality-neutral services 140 run multiple operating system personalities 150 on a variety of hardware platforms.

The host multi-processor 100 shown in Figure 1 includes memory 102 connected by means of a bus 104 to an auxiliary storage 106 which can be for example a disc drive, a read only or a read/write optical storage, or any other bulk storage device. Also connected to the bus 104 is the I/O adaptor 108 which in turn may be connected to a keyboard, a monitor display, a telecommunications adaptor, a local area network adaptor, a modem, multi-media interface devices, or other I/O devices. Also connected to the bus 104 is a first processor A, 110 and a second processor B, 112. The example shown in Figure 1 is of a symmetrical multi-processor configuration wherein the two uni-processors 110 and 112 share a common memory address space 102. Other configurations of single or multiple processors can be shown as equally suitable examples. The processors can be, for example, an Intel 386 (TM) CPU, Intel 486 (TM) CPU, a Pentium (TM) processor, a Power PC (TM) processor, or other uni-processor devices.

The memory 102 includes the microkernel system 115 stored therein, which comprises the microkernel 120, the personality neutral services (PNS) 140, and the personality servers 150. The microkernel system 115 serves as the operating system for the application programs 180 stored in the memory 102.

An objective of the invention is to provide an operating system that behaves like a traditional operating system such as UNIX or OS/2. In other words, the operating system will have the personality of OS/2 or UNIX, or some other traditional operating system.

The microkernel 120 contains a small, message-passing nucleus of system software running in the most privileged state of the host multi-processor 100, that controls the basic operation of the machine. The microkernel system 115 includes the microkernel 120 and a set of servers and device drivers that provide personality neutral services 140. As the name implies, the personality neutral servers and device drivers are not dependent on any personality such as UNIX or OS/2. They depend on the microkernel 120 and upon each other. The personality servers 150 use the message passing services of the microkernel 120 to communicate with the personality neutral services 140. For example, UNIX, OS/2 or any other personality server can send a message to a personality neutral disc driver and ask it to read a block of data from the disc. The disc driver reads the block and returns it in a message. The message system is optimized so that large amounts of data are transferred rapidly by manipulating pointers; the data itself is not copied.

By virtue of its size and ability to support standard programming services and features as application programs, the microkernel 120 is simpler than a standard operating system. The microkernel system 115 is broken down into modular pieces that are configured in a variety of ways, permitting larger systems to be built by adding pieces to the smaller ones. For example, each personality neutral server 140 is logically separate and can be configured in a variety of ways. Each server runs as an application program and can be debugged using application debuggers. Each server runs in a separate task and errors in the server are confined to that task.

Figure 1 shows the microkernel 120 including the interprocess communications module (IPC) 122, the virtual memory module 124, tasks and threads module 126, the host and processor sets 128, I/O support and interrupts 130, and machine dependent code 125.

The personality neutral services 140 shown in Figure 1 includes the multiple personality support 142 which includes the master server, initialization, and naming. It also includes the default pager 144. It also includes the device support 146 which includes multiple personality support and device drivers. It also includes other personality neutral products 148, including a file server, network services, database engines and security.

The personality servers 150 are for example the dominant personality 152 which can be, for example, a UNIX personality. It includes a dominant personality server 154 which would be a UNIX server, and other dominant personality services 155 which would support the UNIX dominant personality. An alternate dominant personality 156 can be for example OS/2. Included in the alternate personality 156 are the alternate personality server 158 which would characterize the OS/2 personality, and other alternate personality services for OS/2, 159.

Dominant personality applications 182 shown in Figure 1, associated with the UNIX dominant personality example, are UNIX-type applications which would run on top of the UNIX operating system personality 152. The alternate personality applications 186 shown in Figure 1, are OS/2 applications which run on top of the OS/2 alternate personality operating system 156.

Figure 1 shows that the Microkernel System 115 carefully splits its implementation into code that is completely portable from processor type to processor type and code that is dependent on the type of processor in the particular machine on which it is executing. It also segregates the code that depends on devices into device drivers; however, the device driver code, while device dependent, is not necessarily dependent on the processor architecture. Using multiple threads per task, it provides an application environment that permits the use of multi-processors without requiring that any particular machine be a multi-processor. On uni-processors, different threads run at different times. All of the support needed for multiple processors is concentrated into the small and simple microkernel 120.

This section provides an overview of the structure of the Microkernel System 115. Later sections describe each component of the structure in detail and describe the technology necessary to build a new program using the services of the Microkernel System 115.

The Microkernel System 115 is a new foundation for operating systems. It provides a comprehensive environment for operating system development with the following features:

Support for multiple personalities
Extensible memory management
Interprocess communication
Multi-threading
Multi-processing

The Microkernel System 115 provides a concise set of kernel services implemented as a pure kernel and an extensive set of services for building operating system personalities implemented as a set of user-level servers. Objectives of the Microkernel System 115 include the following:

Permit multiple operating system personalities to work together in harmony;
Provide common programming for low-level system elements, such as device drivers and file systems;
Exploit parallelism in both operating system and user applications;
Support large, potentially sparse address spaces with flexible memory sharing;
Allow transparent network resource access;
Be compatible with existing software environments, such as OS/2 and UNIX; and
Portable (to 32-bit and 64-bit platforms).

The Microkernel System 115 is based on the following concepts:

User mode tasks performing many traditional operating system functions (for example, file system and network access);
A basic set of user-level run time services for creating operating systems;
A simple, extensible communication kernel;
An object basis with communication channels as object references; and
A client/server programming model, using synchronous and asynchronous inter-process communication.

The basis for the Microkernel System 115 is to provide a simple, extensible communication kernel. It is an objective of the Microkernel System 115 to permit the flexible configuration of services in either user or kernel space with the minimum amount of function in the kernel proper. The kernel must provide other support besides task-to-task communication, including:

Management of points of control (threads);
Resource assignment (tasks);
Support of address spaces for tasks; and
Management of physical resources, such as physical memory, processors, interrupts, DMA channels, and clocks.

User mode tasks implement the policies regarding resource usage. The kernel simply provides mechanisms to enforce those policies. Logically above the kernel is the Personality-Neutral services 140 (PNS) layer. The PNS provide a C runtime environment, including such basic constructs as string functions, and a set of servers which include:

Name Server	- Allows a client to find a server
Master Server	- Allows programs to be loaded and started

Kernel Abstractions

One goal of the Microkernel System 115 is to minimize abstractions provided by the kernel itself, but not to be minimal in the semantics associated with those abstractions. Each of the abstractions provided has a set of semantics associated with it, and a complex set of interactions with the other abstractions. This can make it difficult to identify key ideas. The main kernel abstractions are:

Task - Unit of resource allocation, large access space and port right
Thread - Unit of CPU utilization, lightweight (low overhead)

Port - A communication channel, accessible only through the send/receive capabilities or rights

Message - A collection of data objects

Memory object - The internal unit of memory management (Refer to Section 2, Architectural Model, for a detailed description of the task, thread, port, message and memory object concepts).

Tasks and Threads

The Microkernel System 115 does not provide the traditional concept of process because: All operating system environments have considerable semantics associated with a process (such as user ID, signal state, and so on). It is not the purpose of the microkernel to understand or provide these extended semantics. Many systems equate a process with an execution point of control. Some systems do not. The microkernel 120 supports multiple points of control separately from the operating system environment's process. The microkernel provides the following two concepts:

Task

Thread

(Refer to Section 2, Architectural Model, for a detailed description of the task and thread concepts).

Memory Management

The kernel provides some memory management. Memory is associated with tasks. Memory objects are the means by which tasks take control over memory management. The Microkernel System 115 provides the mechanisms to support large, potentially sparse virtual address spaces. Each task has an associated address map that is maintained by the kernel and controls the translation of virtual address in the task's address space into physical addresses. As in virtual memory systems, the contents of the entire address space of any given task might not be completely resident in physical memory at the same time, and mechanisms must exist to use physical memory as a cache for the virtual address spaces of tasks. Unlike traditional virtual memory designs, the Microkernel System 115 does not implement all of the caching itself. It gives user mode tasks the ability to participate in these mechanisms. The PNS include a user task, the default pager 144, that provides paging services for memory.

Unlike other resources in the Microkernel System 115 ,virtual memory is not referenced using ports. Memory can be referenced only by using virtual addresses as indices into a particular task's address space. The memory and the associated address map that defines a task's address space can be partially shared with other tasks. A task can allocate new ranges of memory within its address space, de-allocate them, and change protections on them. It can also specify inheritance properties for the ranges. A new task is created by specifying an existing task as a base from which to construct the address space for the new task. The inheritance attribute of each range of the memory of the existing task determines whether the new task has that range defined and whether that range is virtually copied or shared with the existing task. Most virtual copy operations for memory are achieved through copy-on-write optimizations. A copy-on-write optimization is accomplished by protected sharing. The two tasks share the memory to be copied, but with read-only access. When either task attempts to modify a portion of the range, that portion is copied at that time. This lazy evaluation of memory copies is an important performance optimization performed by the Microkernel System 115 and important to the communication/memory philosophy of the system.

Any given region of memory is backed by a memory object. A memory manager task provides the policy governing the relationship between the image of a set of pages while cached in memory (the physical memory contents of a memory region) and the image of that set of pages when not cached (the abstract memory object). The PNS has a default memory manager or pager that provides basic non-persistent memory objects that are zero-filled initially and paged against system paging space.

Task to Task Communication

The Microkernel System 115 uses a client/server system structure in which tasks (clients) access services by making requests of other tasks (servers) through messages sent over a communication channel. Since the microkernel 120 provides very few services of its own (for example, it provides no file service), a microkernel 120 task must communicate with many other tasks that provide the required services. The communication channels of the interprocess communication (IPC) mechanism are called ports. (Refer to Section 2, Architectural Model, for a detailed description of a Port). A message is a collection of data, memory regions, and port rights. A port right is a name by which a task, that holds the right, names the port. A task can manipulate a port only if it holds the appropriate port rights. Only one task can hold the receive right for a port. This task is allowed to receive (read) messages from the port queue. Multiple tasks can hold send rights to the port that allow them to send (write) messages into the queue. A task communicates with another task by building a data structure that contains a set of data elements, and then performing a message-send operation on a port for which it holds a send right. At some later time, the task holding the receive right to that port performs a message-receive operation.

Note: This message transfer is an asynchronous operation. The message is logically copied into the receiving task (possibly with copy-on-write optimizations). Multiple threads within the receiving task can be attempting to receive messages from a given port, but only one thread will receive any given message.

Section 2. Architectural Model

The Microkernel System 115 has, as its primary responsibility, the provision of points of control that execute instructions within a framework. These points of control are called threads. Threads execute in a virtual environment. The virtual environment provided by the kernel contains a virtual processor that executes all of the user space accessible hardware instructions, augmented by user-space PNS and emulated instructions (system traps) provided by the kernel. The virtual processor accesses a set of virtualized registers and some virtual memory that otherwise responds as does the machine's physical memory. All other hardware resources are accessible only through special combinations of memory accesses and emulated instructions. Note that all resources provided by the kernel are virtualized. This section describes the top level elements of the virtual environment as seen by threads.

Elements of the Personality Neutral Services (PNS)

The PNS 140 portion of the Microkernel System 115 consists of services built on the underlying microkernel 120. This provides some functions that the kernel itself depends on, as well as a basic set of user-level services for the construction of programs. These programs can serve requests from multiple operating system personality clients and are used to construct the operating system personalities themselves. In addition, there is an ANSI C run time environment for the construction of PNS programs in standard C and some supplemental functions that have definitions taken from the POSIX standard. Besides the libraries that define the PNS themselves, there are many libraries that exist within the PNS that are a part of the microkernel proper. These libraries represent the interfaces that the microkernel exports and the support logic for the Message Interface Generator (MIG) which is used with the Microkernel System's 115 interprocess communications facilities. The structure of the PNS environment library hides the details of the implementation of each service from its callers. Some libraries, such as one of the C run time libraries, implement all of their functions as local routines that are loaded into the address space of the caller while other libraries consist of stubs that invoke the microkernel's IPC system to send messages to servers. This architecture permits the flexible implementation of function: servers can be replaced by other servers and services can be combined into single tasks without affecting the sources of the programs that use them. A key element of the PNS environment is that, it does not constitute a complete operating system. Instead, the PNS depend on the existence of a personality. The dominant personality 152, that is loaded first during system start-up, is the operating system personality which provides the user interface on the system and provides services to its clients and to elements of the PNS. Thus, the dominant personality is a server of "last resort". The dominant personality implements whatever services are defined by the PNS libraries but are not implemented by another server.

The microkernel 120 is also dependent on some elements of the PNS. There are cases when it sends messages to personality-neutral servers to complete internal kernel operations. For example, in resolving a page fault, the microkernel 120 may send a message to the default pager 144. The default pager 144 then reads in the page that the kernel needs from a hard disk. Although the page fault is usually being resolved on behalf of a user task, the kernel is the sender of the message.

Run Time

The PNS run time provides a set of ANSI C and POSIX libraries that are used to support a standard C programming environment for programs executing in this environment. The facilities include typical C language constructs. Like all systems, the microkernel system 115 has, as its primary responsibility, the provision of points of control that execute instructions within a framework. In the microkernel 120, points of control are called threads. Threads execute in a virtual environment. The virtual environment provided by the microkernel 120 consists of a virtual processor that executes all of the user space accessible hardware instructions, augmented by emulated instructions (system traps) provided by the kernel; the virtual processor accesses a set of virtualized registers and some virtual memory that otherwise responds as does the machine's physical memory. All other hardware resources are accessible only through special combinations of memory accesses and emulated instructions. Note that all resources provided by the microkernel are virtualized. This section describes the top level elements of the virtual environment seen by the microkernel threads.

Elements of the Kernel

The microkernel 120 provides an environment consisting of the elements described in the following list of Kernel Elements:

Thread: An execution point of control. A thread is a lightweight entity. Most of the state pertinent to a thread is associated with its containing task.

Task: A container to hold references to resources in the form of a port name space, a virtual address space, and a set of threads.

Security Token: A security feature passed from the task to server, which performs access validations.

Port: A unidirectional communication channel between tasks.

Port Set: A set of ports which can be treated as a single unit when receiving a message.

Port Right: Allows specific rights to access a port.

Port Name Space: An indexed collection of port names that names a particular port right.

Message A collection of data, memory regions and port rights passed between two tasks.

Message Queue: A queue of messages associated with a single port.

Virtual Address Space: A sparsely populated, indexed set of memory pages that can be referenced by the threads within a task. Ranges of pages might have arbitrary attributes and semantics associated with them through mechanisms implemented by the kernel and external memory managers.

Abstract Memory Object: An abstract object that represents the non-resident state of the memory ranges backed by this object. The task that implements this object is called a memory manager. The abstract memory object port is the port through which the kernel requests action of the memory manager.

Memory Object Representative: The abstract representation of a memory object provided by the memory manager to clients of the memory object. The representative names the associated abstract memory object and limits the potential access modes permitted to the client.

Memory Cache Object: A kernel object that contains the resident state of the memory ranges backed by an abstract memory object. It is through this object that the memory manager manipulates the clients' visible memory image.

Processor: A physical processor capable of executing threads.

Processor Set: A set of processors, each of which can be used to execute the threads assigned to the processor set.

Host: The multiprocessor as a whole.

Clock: A representation of the passage of time. A time value incremented at a constant frequency.

Many of these elements are kernel implemented resources that can be directly manipulated by threads. Each of these elements are discussed in detail in the paragraphs that follow. However, since some of their definitions depend on the definitions of others, some of the key concepts are discussed in simplified form so that a full discussion can be understood.

Threads

A thread is a lightweight entity. It is inexpensive to create and requires low overhead to operate. A thread has little state (mostly its register state). Its owning task bears the burden of resource management. On a multiprocessor it is possible for multiple threads in a task to execute in parallel. Even when parallelism is not the goal, multiple threads have an advantage because each thread can use a synchronous programming style, instead of asynchronous programming with a single thread attempting to provide multiple services. A thread contains the following features:

1. a point of control flow in a task or a stream of instruction execution ;

2. access to all of the elements of the containing task;

3. executes in parallel with other threads, even threads within the same task; and

4. minimal state for low overhead.

A thread is the basic computational entity. A thread belongs to only one task that defines its virtual address space. To affect the structure of the address space, or to reference any resource other than the address space, the thread must execute a special trap instruction. This causes the kernel to perform operations on behalf of the thread, or to send a message to an agent on behalf of the thread. These traps manipulate resources associated with the task containing the thread. Requests can be made of the kernel to manipulate these entities: to create and delete them and affect their state. The kernel is a manager that provides resources (such as those listed above) and services. Tasks may also provide services, and implement abstract resources. The kernel provides communication methods that allow a client task to request that a server task (actually, a thread executing within it) provide a service. In this way, a task has a dual identity. One identity is that of a resource managed by the kernel, whose resource manager executes within the kernel. The second identity is that of a supplier of resources for which the resource manager is the task itself.

A thread has the following state:

1. Its machine state (registers, etc.), which change as the thread executes and which can also be changed by a holder of the kernel thread port;

2. A small set of thread specific port rights, identifying the thread's kernel port and ports used to send exception messages on behalf of the thread;

3. A suspend count, non-zero if the thread is not to execute instructions; and

4. Resource scheduling parameters.

A thread operates by executing instructions in the usual way. Various special instructions trap to the kernel, to perform operations on behalf of the thread. The most important of these kernel traps is the mach_msg_trap. This trap allows the thread to send messages to the kernel and other servers to operate upon resources. This trap is almost never directly called; it is invoked through the mach_msg library routine. Exceptional conditions, such as "floating point overflow" and "page not resident", that arise during the thread's execution, are handled by sending messages to a port. The port used depends on the nature of the condition. The outcome of the exceptional condition is determined by setting the thread's state and/or responding to the exception message. The following operations can be performed on a thread:

Creation and destruction;
Suspension and resumption (manipulating the suspend count);
Machine state manipulation Special port (such as exception; port) manipulation; and
Resource (scheduling) control.

Tasks

A task is a collection of system resources. These resources, with the exception of the address space, are referenced by ports. These resources can be shared with other tasks if rights to the ports are so distributed. Tasks provide a large, potentially sparse address space, referenced by machine address. Portions of this space can be shared through inheritance or external memory management. Note: A task has no life of its own. It contains threads which execute instructions. When it is said "a task Y does X" what is meant is "a thread contained within task Y does X". A task is an expensive entity. All of the threads in a task share everything. Two tasks share nothing without explicit action, although the action is often simple. Some resources such as port receive rights cannot be shared between two tasks. A task can be viewed as a container that holds a set of threads. It contains default values to be applied to its containing threads. Most importantly, it contains those elements that its containing threads need to execute, namely, a port name space and a virtual address space.

The state associated with a task is as follows:

The set of contained threads;

The associated virtual address space;

The associated port name space, naming a set of port rights, and a related set of port notification requests;

A security token to be sent with messages from the task;

A small set of task specific ports, identifying the task's kernel port, default ports to use for exception handling for contained threads, and bootstrap ports to name other services;

A suspend count, non-zero if no contained threads are to execute instructions;

Default scheduling parameters for threads; and

Various statistics, including statistical PC samples.

Tasks are created by specifying a prototype task which specifies the host on which the new task is created, and which can supply by inheritance various portions of its address space.

The following operations can be performed on a task:

Creation and destruction

Setting the security token

Suspension and resumption

Special port manipulation

Manipulation of contained threads

Manipulation of the scheduling parameters

Security Port

All tasks are tagged with a security token, an identifier that is opaque from the kernel's point of view. It encodes the identity and other security attributes of the task. This security token is included as an implicit value in all messages sent by the task. Trusted servers can use this sent token as an indication of the sender's identity for use in making access mediation decisions. A task inherits the security token of its parent. Because this token is to be used as an unforgeable indication of identity, privilege is required to change this token. This privilege is indicated by presenting the host security port.

A reserved value indicates the kernel's identity. All messages from the kernel carry the kernel identity, except exception messages, which carry the excepting task's identity.

Port

A port is a unidirectional communication channel between a client that requests a service and a server that provides the service. A port has a single receiver and potentially multiple senders. The state associated with a port is as follows:

Its associated message queue

A count of references (rights) to the port

Settable limits on the amount of virtual copy memory and port rights that can be sent in a message through the port.

Kernel services exist to allocate ports. All system entities other than virtual memory ranges are named by ports; ports are also created implicitly when these entities are created. The kernel provides notification messages upon the death of a port upon request. With the exception of the task's virtual address space, all other system resources are accessed through a level of indirection known as a port. A port is a unidirectional communication channel between a client who requests service and a server who provides the service. If a reply is to be provided to such a service request, a second port must be used. The service to be provided is determined by the manager that receives the message sent over the port. It follows that the receiver for ports associated with kernel provided entities is the kernel. The receiver for ports associated with task provided entities is the task providing that entity. For ports that name task provided entities, it is possible to change the receiver of messages for that port to a different task. A single task might have multiple ports that refer to resources it supports. Any given entity can have multiple ports that represent it, each implying different sets of permissible operations. For example, many entities have a name port and a control port that is sometimes called the privileged port. Access to the control port allows the entity to be manipulated. Access to the name port simply names the entity, for example, to return information. There is no system-wide name space for ports. A thread can access only the ports known to its containing task. A task holds a set of port rights, each of which names a (not necessarily distinct) port and which specifies the rights permitted for that port. Port rights can be transmitted in messages. This is how a task gets port rights. A port right is named with a port name, which is an integer chosen by the kernel that is meaningful only within the context (port name space) of the task holding that right. Most operations in the system consist of sending a message to a port that names a manager for the object being manipulated. In this document, this is shown in the form:

object -> function

which means that the function is invoked (by sending an appropriate message) to a port that names the object. Since a message must be sent to a port (right), this operation has an object basis. Some operations require two objects, such as binding a thread to a processor set. These operations show the objects separated by commas. Not all entities are named by ports, and this is not a pure object model. The two main non-port-right named entities are port names/rights themselves, and ranges of memory. Event objects are also named by task local IDs. To manipulate a memory range, a message is sent to the containing virtual address space named by the owning task. To manipulate a port name/right, and often, the associated port, a message is sent to the containing port name space named by the owning task. A subscript notation,

object [id] -> function

is used here to show that an id is required as a parameter in the message to indicate which range or element of object is to be manipulated. The parenthetic notation,

object (port) -> function

is used here to show that a privileged port, such as the host control port, is required as a parameter in the message to indicate sufficient privilege to manipulate the object in the particular way.

Port Sets

A port set is a set of ports that can be treated as a single unit when receiving a message. A mach_msg receive operation is allowed against a port name that either names a receive right, or a port set. A port set contains a collection of receive rights. When a receive operation is performed against a port set, a message is received from one of the ports in the set. The received message indicates from which member port it was received. It is not allowed to directly receive a message from a port that is a member of a port set. There is no concept of priority for the ports in a port set; there is no control provided over the kernel's choice of the port within the port set from which any given message is received.

Operations supported for port sets include:

Creation and deletion
Membership changes and membership queries

Port Rights

A port can only be accessed by using a port right. A port right allows access to a specific port in a specific way. There are three types of port rights as follow:

receive right - Allows the holder to receive messages from the associated port.

send right - Allows the holder to send messages to the associated port.

send-once right - Allows the holder to send a single message to the associated port. The port right self-destructs after the message is sent.

Port rights can be copied and moved between tasks using various options in the mach_msg call, and also by explicit command. Other than message operations, port rights can be manipulated only as members of a port name space. Port rights are created implicitly when any other system entity is created, and explicitly using explicit port creation. The kernel will, upon request, provide notification to a port of one's choosing when there are no more send rights to a port. Also, the destruction of a send-once right (other than by using it to send a message) generates a send-once notification sent to the corresponding port. Upon request, the kernel provides notification of the destruction of a receive right.

Port Name Space

Ports and port rights do not have system-wide names that allow arbitrary ports or rights to be manipulated directly. Ports can be manipulated only through port rights, and port rights can be manipulated only when they are contained within a port name space. A port right is specified by a port name which is an index into a port name space. Each task has associated with it a single port name space. An entry in a port name space can have the following four possible values:

MACH_PORT_NULL - No associated port right.

MACH_PORT_DEAD - A right was associated with this name, but the port to which the right referred has been destroyed.

A port right - A send-once, send or receive right for a port. A port set name - A name which acts like a receive right, but that allows receiving from multiple ports.

Acquiring a new right in a task generates a new port name. As port rights are manipulated by referring to their port names, the port names are sometimes themselves manipulated. All send and receive rights to a given port in a given port name space have the same port name. Each send-once right to a given port have a different port name from any other and from the port name used for any send or receive rights held. Operations supported for port names include the following:

Creation (implicit in creation of a right) and deletion
Query of the associated type
Rename
Upon request, the kernel provides notification of a name becoming unusable.

Since port name spaces are bound to tasks, they are created and destroyed with their owning task.

Message

A message is a collection of data, memory regions and port rights passed between two entities. A message is not a system object in its own right. However, since messages are queued, they are significant because they can hold state between the time a message is sent and when it is received. This state consists of the following:

Pure data
Copies of memory ranges
Port rights
Sender's security token

Message Queues

A port consists of a queue of messages. This queue is manipulated only through message operations (mach_msg) that transmit messages. The state associated with a queue is the ordered set of messages queued, and settable limit on the number of messages.

Virtual Address Space

A virtual address space defines the set of valid virtual addresses that a thread executing within the task owning the virtual address space is allowed to reference. A virtual address space is named by its owning task. A virtual address space consists of a sparsely populated indexed set of pages. The attributes of individual pages can be set as desired. For efficiency, the kernel groups virtually contiguous sets of pages that have the same attributes into internal memory regions. The kernel is free to split or merge memory regions as desired. System mechanisms are sensitive to the identities of memory regions, but most user accesses are not so affected, and can span memory regions freely. A given memory range can have distinct semantics associated with it through the actions of a memory manager. When a new memory range is established in a virtual address space, an abstract memory object is specified, possibly by default, that represents the semantics of the memory range, by being associated with a task (a memory manager) that provides those semantics. A virtual address space is created when a task is created, and destroyed when the task is destroyed. The initial contents of the address space is determined from various options to the task_create call, as well as the inheritance properties of the memory ranges of the prototype task used in that of call. Most operations upon a virtual address space name a memory range within the address space. These operations include the following:

Creating or allocating, and de-allocating a range
Copying a range
Setting special attributes, including "wiring" the page into physical memory to prevent eviction
Setting memory protection attributes
Setting inheritance properties
Directly reading and writing ranges
Forcing a range flush to backing storage
Reserving a range (preventing random allocation within the range)

Abstract Memory Object

The microkernel allows user mode tasks to provide the semantics associated with referencing portions of a virtual address space. It does this by allowing the specification of an abstract memory object that represents the non-resident state of the memory ranges backed by this memory object. The task that implements this memory object and responds to messages sent to the port that names the memory object is called a memory manager. The kernel should be viewed as using main memory as a directly accessible cache for the contents of the various memory objects. The kernel is involved in an asynchronous dialog with the various memory managers to maintain this cache, filling and flushing this cache as the kernel desires, by sending messages to the abstract memory object ports. The operations upon abstract memory objects include the following:

Initialization
Page reads
Page writes
Synchronization with force and flush operations
Requests for permission to access pages
Page copies
Termination

Memory Object Representative

The abstract memory object port is used by the kernel to request access to the backing storage for a memory object. Because of the protected nature of this dialog, memory managers do not typically give access to the abstract memory object port to clients. Instead, clients are given access to memory object representatives. A memory object representative is the client's representation of a memory object. There is only one operation permitted against such a port and that is to map the associated memory object into a task's address space. Making such a request initiates a protocol between the mapping kernel and the memory manager to initialize the underlying abstract memory object. It is through this special protocol that the kernel is informed of the abstract memory object represented by the representative, as well as the set of access modes permitted by the representative.

Memory Cache Object

The portion of the kernel's main memory cache that contains the resident pages associated with a given abstract memory object is referred to as the memory cache object. The memory manager for a memory object holds send rights to the kernel's memory cache object. The memory manager is involved in an asynchronous dialog with the kernel to provide the abstraction of its abstract memory object by sending messages to the associated memory cache object. The operations upon memory cache objects include the following:

Set operational attributes
Return attributes
Supply pages to the kernel
Indicate that pages requested by the kernel are not available
Indicate that pages requested by the kernel should be filled by the kernel's default rules Force delayed copies of the object to be completed
Indicate that pages sent to the memory manager have been disposed
Restrict access to memory pages
Provide performance hints
Terminate

Processor

Each physical processor that is capable of executing threads is named by a processor control port. Although significant in that they perform the real work, processors are not very significant in the microkernel, other than as members of a processor set. It is a processor set that forms the basis for the pool of processors used to schedule a set of threads, and that has scheduling attributes associated with it. The operations supported for processors include the following:

Assignment to a processor set
Machine control, such as start and stop

Processor Set

Processors are grouped into processor sets. A processor set forms a pool of processors used to schedule the threads assigned to that processor set. A processor set exists as a basis to uniformly control the schedulability of a set of threads. The concept also provides a way to perform coarse allocation of processors to given activities in the system. The operations supported upon processor sets include the following:

Creation and deletion
Assignment of processors

Assignment of threads and tasks
Scheduling control

Host

Each machine (uniprocessor or multiprocessor) in a networked microkernel system runs its own instantiation of the microkernel. The host multiprocessor 100 is not generally manipulated by client tasks. But, since each host does carry its own microkernel 120, each with its own port space, physical memory and other resources, the executing host is visible and sometimes manipulated directly. Also, each host generates its own statistics. Hosts are named by a name port which is freely distributed and which can be used to obtain information about the host and a control port which is closely held and which can be used to manipulate the host. Operations supported by hosts include the following:

Clock manipulation
Statistics gathering
Re-boot
Setting the default memory manager
Obtaining lists of processors and processor sets

Clock

A clock provides a representation of the passage of time by incrementing a time value counter at a constant frequency. Each host or node in a multicomputer implements its own set of clocks based upon the various clocks and timers supported by the hardware as well as abstract clocks built upon these timers. The set of clocks implemented by a given system is set at configuration time. Each clock is named by both a name and a control or privileged port. The control port allows the time and resolution of the clock to be set. Given the name port, a task can perform the following:

Determine the time and resolution of the clock.
Generate a memory object that maps the time value.
Sleep (delay) until a given time.
Request a notification or alarm at a given time.

Section 3. Tasks and Threads

This section discusses the user visible view of threads and tasks. Threads are the active entities in the Microkernel System 115. They act as points of control within a task, which provides them with a virtual address space and a port name space with which other resources are accessed.

Threads

A thread is the basic computational entity. A thread belongs to only one task that defines its virtual address space. A thread is a lightweight entity with a minimum of state. A thread executes in the way dictated by the hardware, fetching instructions from its task's address space based on the thread's register values. The only actions a thread can take directly are to execute instructions that manipulate its registers and read and write into its memory space. An attempt to execute privileged machine instructions, though, causes an exception. The exception is discussed later. To affect the structure of the address space, or to reference any resource other than the address space, the thread must execute a special trap instruction which causes the kernel to perform operations on behalf of the thread, or to send a message to some agent on behalf of the thread. Also, faults or other illegal instruction behavior cause the kernel to invoke its exception processing.

Figure 2. shows the client visible structure associated with a thread. The thread object is the receiver for messages sent to the kernel thread port. Aside from any random task that holds a send right for this thread port, the thread port is also accessible as the thread's thread self port, through the containing processor set or the containing task.

Reference is made here to the above cited copending United States Patent Application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, entitled "METHOD AND APPARATUS FOR MANAGEMENT OF MAPPED AND UNMAPPED REGIONS OF MEMORY IN A MICROKERNEL DATA PROCESSING SYSTEM", which is incorporated herein by reference for its more detailed discussion of this topic.

Tasks

A task can be viewed as a container that holds a set of threads. It contains default values to be applied to its containing threads. Most importantly, it contains those elements that its containing threads need to execute, namely, a port name space and a virtual address space.

Figure 3. shows the client visible task structures. The task object is the receiver for messages sent to the kernel task port. Aside from any random task that may hold a send right to the task port, the task port can be derived from the task's task self port, the contained threads or the containing processor set.

Reference is made here to the above cited copending United States Patent Application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, entitled "METHOD AND APPARATUS FOR MANAGEMENT OF MAPPED AND UNMAPPED REGIONS OF MEMORY IN A MICROKERNEL DATA PROCESSING SYSTEM", which is incorporated herein by reference for its more detailed discussion of this topic.

Section 4. IPC

With the exception of its shared memory, a microkernel task interacts with its environment purely by sending messages and receiving replies. These messages are sent using ports. A port is a communication channel that has a single receiver and can have multiple senders. A task holds rights to these ports that specify its ability to send or receive messages.

Ports

A port is a unidirectional communication channel between a client who requests a service and a server who provides the service. A port has a single receiver and can have multiple senders. A port that represents a kernel supported resource has the kernel as the receiver. A port that names a service provided by a task has that task as the port's receiver. This receivership can change if desired, as discussed under port rights. The state associated with a port is:

The associated message queue
A count of references or rights to the port
Port right and out-of-line memory receive limits
Message sequence number
Number of send rights created from receive right
Containing port set
Name of no-more-sender port if specified

Figure 4 shows a typical port, illustrating a series of send rights and the single receive right. The associated message queue has a series of ordered messages. One of the messages is shown in detail, showing its destination port, reply port reference, a send- and-receive right being passed in the message, as well as some out-of-line or virtual copy memory.

Few operations affect the port itself. Most operations affect port rights or a port name space containing those rights, or affect the message queue. Ports are created implicitly when any other system entity is created. Also, mach_reply_port creates a port. Ports are created explicitly by port_name_space [port_name] → mach_port_allocate and port_name_space [port_name]→ mach_port_allocate_name. A port cannot be explicitly destroyed. It is destroyed only when the receive right is destroyed.

The attributes of a port are assigned at the time of creation. Some of these attributes, such as the limit on the number of port rights or amount of out-of-line memory that can be received in a message can be changed with port_name_space [port_name] → mach_port_set_attributes. These attributes can be obtained with port_name_space [port_name]→ mach_port_get_attributes.

The existence of ports is of obvious importance to all involved. As such, many tasks using a port may wish to be notified, through a message, when the port dies. Such notifications are requested with an option to mach_msg (MACH_RCV_NOTIFY), as well as with port_name_space [port_name]→ mach_port_request_notification. The resultant dead name notification indicates that a task's port name has gone dead because of the destruction of the named port. The message indicates the task's name for the now dead port. (This is discussed under port name spaces.).

Messages

A message is a collection of data, out-of-line memory regions, and port rights passed between two entities. A message is not a manipulable system object in its own right. However, because messages are queued, they are significant because they can hold state between the time a message is sent and the time it is received. Besides pure data, a message can also contain port rights. This is significant. In this way a task obtains new rights, by receiving them in a message. A message consists of an Interprocess Communication (IPC) subsystem parsed control section and a data section. In addition, the message may point to regions of data to be transferred which lie outside the message proper. These regions may contain port rights (out of line port arrays). A message also carries the sender's security token. The control section of a message is made up of a header and optionally a message body and one or more message descriptors. The header specifies a port name for the port to which the message is sent and an auxiliary port name for the port to which a reply is to be sent if a reply is requested.

The message body, when present, follows the header and declares the number of descriptors to follow. If there are no descriptors, the message is not considered "complex", meaning no overt translation or copying of data is required of the IPC subsystem. Non "complex" messages do not contain a message body. Each descriptor describes a section of kernel manipulated data such as an out-of-line memory region, port right, and port right array. The data contained in the message data section is treated as an anonymous array of bytes by the IPC subsystem. Both the sender and receiver of the message must share a common understanding of the data format. For messages that originate in a Message Interface Generator (MIG) generated routine, the first eight bytes of data contain machine encoding information for possible intermachine conversion of data contained in the message.

A descriptor for a single port right names the right as well as any special transformations performed by mach_msg, such as moving the right instead of making a copy or generating a right from a right of a different type. A descriptor for an "out-of-line" port array also specifies the IPC processing for the set of rights, which must all be of the same type, but specifies the address and size of an array of port names. An out-of-line data descriptor specifies the size and address of the out-of-line region. The region need not start at the beginning of a page nor contain an integral number of pages. Only the specified data is logically transmitted. The descriptor can specify that the act of queuing the message will de-allocate the memory range, as if by vm_deallocate, from the sending task. The sender is given the choice of sending a physical or virtual copy of the data. Requesting a virtual copy permits the kernel to use its virtual copy, that is, copy-on-write, mechanisms to efficiently copy large amounts of data. As a result, the receiver may see a virtual copy that is backed by a sender memory manager. Both sender and receiver might experience indeterminacy in access time to the memory because of the potential virtual copy mechanisms used. The choice of a physical copy guarantees deterministic access by the sender and receiver to the data. Neither the values in the out-of-line data regions or in the message data are typed. Only the receiver of a message, with the knowledge of the port over which it came and the message data, can interpret and possibly transform these data areas.

Message Trailers

When a message is received, a trailer is appended to the end by default. The trailer is generally added by the IPC subsystem and is physically contiguous with the message. The trailer contains various transmission-related fields, such as message sequence number and sender security token. The format and size of the trailer are controllable via receive side options to mach_msg. It is also possible as an option for the sender to supply a trailer.

Message Queues

A port consists of a queue of messages. The queue is manipulated only through message operations (mach_msg) that transmit messages. The only controllable state for a message queue is its size. This can be set when port_name_space [port_name] → mach_port_set_attributes is given the receive right for the associated port. If a message queue is full, no more messages can be queued. The callers will block. Messages sent to a port are delivered reliably. Reception of a message guarantees that all previous messages were received and that they were received in the order in which they were queued on the receive port.

Port Rights

A port right is an entity that indicates the right to access a specific port in a specific way. A port can only be accessed through a port right. In this context, there are three types of port rights as follows:

receive right - Allows the holder to receive messages from the associated port.

send right - Allows the holder to send messages to the associated port.

send-once right - Allows the holder to send a single message to the associated port. The right self-destructs after the message is sent.

Port rights are a secure, location independent way of identifying ports. These rights are kernel protected entities. Clients manipulate port rights only through port names they have to these rights.

Basic Manipulation

mach_msg is one of the principal ways that rights are manipulated. Port rights can be moved between tasks, that is, deleted from the sender and added to the receiver in messages. Option flags in a message can cause mach_msg to make a copy of an existing send right, or to generate a send or a send-once right from a receive right. Rights can also be forcefully copied or moved by port_name_space [port_name]→ mach_port_extract_right (the equivalent of the target sending the right in a message) and port_name_space [port_name]→ mach_port_insert_right (the equivalent of the target receiving the right in a message). Besides message operations, port rights can be manipulated only as members of a port name space.

Figure 5. shows a series of port rights, contained in a port name space or in transit in a message. A port set is also shown in the port name space. Port rights are created implicitly when any other system entity is created. Mach_reply_port creates a port right. Port rights are created explicitly by port_name_space [port_name] → mach_port_allocate and port_name_space [port_name] → mach_port_allocate_name. A port right is destroyed by port_name_space [port_name]→ mach_port_deallocate and port_name_space [port_name]→ mach_port_destroy. Destruction can also be a by-product of port name space manipulations, such as by port_name_space [port_name]→ mach_port_mod_refs. Given a receive right, some status information can be obtained with port_name_space [port_name]→ mach_port_get_attributes.

No-More-Senders Notification

The system maintains a system-wide count of the number of send and send-once rights for each port. This includes rights in transit in messages, including the destination and reply port rights. The receiver of a port may wish to know if there are no more send rights for the port, indicating that the port may no longer have value. A notification of this form can be requested using port_name_space [port_name]→ mach_port_request_notification. This notification depends on the concept of a make-send count, discussed as a part of port name spaces. The movement to another task of the receive right cancels the outstanding no-more-senders notification request, and sends a send-once notification to indicate this cancellation. No-more-senders notification occurs when the number of existing send rights goes to zero without regards to the number of outstanding send-once rights.

Send-Once Rights

A send-once right allows a single message to be sent. These rights are generated only from the receive right. A send-once right has the property that it guarantees that a message will result from it. In the normal case, a send-once right is consumed by using it as the destination port in a message. The right is silently destroyed when the message is received. The send-once right can be moved from task to task when it is not being used as a destination right, until such time as it is consumed. If the right is destroyed in any way other than by using it to send a message, a send-once notification is sent to the port. Most of the ways in which a send-once right can be destroyed, when it is not used, are fairly obvious. There are two obscure cases:

The send-once right was specified as the target for a no-senders notification and the port for which the no-senders notification was requested is deleted or the receive right moved. Since there will be no forth coming no-senders notification, a send-once notification is generated instead.

In the process of performing a message receive, the task gives away its receive right after the message is de-queued from the port but prior to its being returned to the task (refer to the details of message transmission below). A send-once notification is sent to the destination port signifying the lost association between the message sent via the send- once right and the port.

Port Name Space

Ports are not named, but port rights are. A port right can only be named by being contained within a port name space. A port is specified by a port name which is an index into a port name space. Each task is associated with a port name space.

An entry in a port name space can have four possible values:

MACH_PORT_NULL - No associated port right.

MACH_PORT_DEAD - A right was associated with this name, but the port to which the right referred is now dead. The port name is kept in this state until explicit action is taken to avoid reusing this name before the client task understands what happened. a port right - A send-once, send or receive right for a port. a port set name - A name which acts like a receive right, but that allows receiving from multiple ports. This is discussed in the next section.

Each distinct right that a port name space contains does not necessarily have a distinct name in the port name space. Send-once rights always consume a separate name for each distinct right. Receive and send rights to the same port coalesce. That is, if a port name space holds three send rights for a port, it will have a single name for all three rights. A port name has an associated reference count for each type of right such as send-once, send, receive, port set, and dead name, that is associated with the name. If the port name space also holds the receive right, that receive right will have the same name as the send right.

A name becomes dead when its associated port is destroyed. It follows that a task holding a dead name cannot be holding a receive right under that name as well. The dead name only has a non-zero reference count for the number of send references previously held by that name. A task can be notified with a message sent to it, when one of its names becomes dead using port_name_space [port_name]→ mach_port_request_notification. Receiving this notification message increments the reference count for the dead name, to avoid a race with any threads manipulating the name.

Whenever a task acquires a right, it is assigned a port name subject to the above rules. Acquiring a right increments the name's reference count for the type of that right. The reference count can be obtained with port_name_space [port_name]→ mach_port_get_refs.

Although a port name can be explicitly destroyed and all references removed using port_name_space [port_name] → mach_port_destroy, port names are typically manipulated by modifying the user reference count. port_name_space [port_name]→ mach_port_mod_refs modifies the reference count for a specified right type associated with a name. port_name_space [port_name]→ mach_port_deallocate is similar to mach_port_mod_refs, but it always decrements the count by 1, and it will only decrement the send or send-once reference count. This routine is useful for manipulating the reference count for a port name that may have become dead since the decision was made to modify the name. Options to mach_msg that actually move a right and also port_name_space [port_name]→ mach_port_extract_right can cause the name's reference count to be decremented. Port names are freed when all the reference counts go to zero. If a port name is freed and a dead-name notification is in effect for the name, a port- deleted notification is generated. A name with a dead-name notification request in effect can be in only one of three states:

Naming a valid right

MACH_PORT_DEAD, with a dead-name notification having been sent when the name became dead

MACH_PORT_NULL, with a port-deleted notification having been sent when the name became null

Information about a name, such as, the type of the name, can be obtained with port_name_space [port_name]→ mach_port_type. The list of assigned names is obtained with port_name_space [port_name]→ mach_port_names. The name by which a right is known can be changed with port_name_space [port_name]→mach_port_rename. Given a receive right name, some status information can be obtained, with port_name_space [port_name]→ mach_port_get_attributes.

Port names that name receive rights have an associated make-send count, used for no- more-sender notification processing. The make-send count is the kernel's count of the number of times a send right was made from the receive right (with a message element that is a port right specifying a MACH_MSG_TYPE_MAKE_SEND type descriptor for mach_msg). This make-send count is set to zero when a port is created, and reset to zero whenever the receive right is transmitted in a message. It can also be changed with port_name_space [port_name]→ mach_port_set_mscount. The make-send count is included in the no-more-senders notification message. Note that a no-senders notification indicates the lack of send rights at the time the notification was generated There may still be outstanding send-once rights. A task can easily keep track of the send-once rights since every send-once right guarantees a message or send-once notification. Received messages are stamped with a sequence number, taken from the port from which the message was received. Messages received from a port set are stamped with a sequence number from the appropriate member port. Sequence numbers placed into sent messages are overwritten. Newly created ports start with a zero sequence number, and the sequence number is reset to zero whenever the port's receive right is moved. It can also be set explicitly with port_name_space [port_name]→ mach_port_set_seqno. When a message is de-queued from the port, it is stamped with the port's sequence number and the port's sequence number is then incremented. The de-queue and increment operations are atomic, so that multiple threads receiving messages from a port can use the msgh_seqno field to reconstruct the original order of the messages.

Since port name spaces are bound to tasks, they are created and destroyed with their owning task.

Port Sets

A port set is a set of ports that can be treated as a single unit when receiving a message. A mach_msg receive operation is allowed against a port name that either names a receive right, or a port set. A port set contains a collection of receive rights. When a receive operation is performed against a port set, a message will be received at random from one of the ports in the set. The order of delivery of messages on a port set is indeterminate and subject to implementation with the following two caveats:

1) No member of a port set shall suffer resource starvation.

2) The order of arrival of a message with respect to other messages on the same port is preserved.

Each of the receive rights in the set has its own name, and the set has its own name. A receive against a port set reports the name of the receive right whose port provided the message. A receive right can belong to only one port set. A task may not directly receive from a receive right that is in a port set. A port set is created with port_name_space [port_name]→ mach_port_allocate or port_name_space [port_name]→ mach_port_allocate_name. It is destroyed by port_name_space [port_name]→ mach_port_destroy or port_name_space [port_name]→ mach_port_deallocate. Manipulations of port sets is done with port_name_space [port_name]→ mach_port_move_member. This call can add a member to a set, remove it from a set, or move it from one set to another. The membership of a port set can be found with port_name_space [port_name]→ mach_port_get_set_status.

Message Transmission

The mach_msg system call sends and receives microkernel messages. For a message to actually be transferred between two tasks, the sender must make a mach_msg call with the send option and the proper target port. The receiver must make a mach_msg call with the receive option, undertaking a receive on the target port. The order of the two calls is unimportant.

The send operation queues a message to a port. The message carries a copy of the caller's data. The caller can freely modify the message buffer and out-of-line regions after the send operation returns without affecting the data sent. Data specified as out-of- line in the message is passed as a virtual copy or a physical copy depending on sender options and the kernel's choice of mechanisms. The kernel constructs a virtual memory image of the set of pages that define the region. "Out-of-line" port arrays are physically copied and translated to port names appropriate to the destination space. If the kernel constructs a virtual copy, it zeroes the portion of the first page preceding the data in the virtual copy and the portion of the last page following the data in the virtual copy. The caller blocks until the message can be queued, unless one of the following happens:

The message is being sent to a send-once right. These messages always forcibly queue.

The mach_msg operation is aborted (thread_abort). By default, the mach_msg library routine retries operations that are interrupted.

The send operation exceeds its time-out value.

The port is destroyed.

Sending a message is a two-step process. The first step involves constructing a kernel copy of the message. The second step involves the queuing of the message. Failures during the first step, such as invalid port rights or data addresses, cause the message send to fail with an error return, but with no ill effect. Failures during the second step may also occur when the send time-out value is exceeded or an interrupt (thread_abort) occurs. These failures also cause the send to fail, but in these situations the kernel tries to return the message contents to the caller with a pseudo-receive operation. This pseudo-receive operation prevents the loss of port rights or memory that only exist in the message (for example, a receive right that was moved into the message, or out-of-line memory sent with the de-allocate flag).

The pseudo-receive operation is very similar to a normal receive operation. The pseudo- receive handles the port rights in the message header as if they were in the message body. After the pseudo-receive, the message is ready to be resent. Notice that if the message is not resent, out-of-line memory ranges may have moved and some port rights may have changed names.

The receive operation de-queues a message from a port. The receiving task acquires the port rights and out-of-line memory ranges carried in the message. The caller must supply a buffer into which the header and body will be copied. The format of the received message is the same as when sent. If the message does not fit, it is destroyed. An option (MACH_RCV_LARGE) allows the caller to receive an error instead, along with the buffer size that would be needed, so that another receive operation can be attempted with an appropriate sized buffer.

A received message can contain port rights and out-of-line memory. Received port rights and memory should be consumed or de-allocated in some fashion. Resource shortages that prevent the reception of a port right or out-of-line memory destroy that entity. The reception of port rights follows the same rules as the insertion of a port right by any other means as described under port name spaces. The port right descriptors describe the type of rights received.

For each out-of-line region transmitted, the kernel returns a descriptor in the received message which locates the memory and indicates whether the kernel has chosen (or has been directed by the sender) to send a physical or virtual copy. Sending a physical copy guarantees deterministic behavior. Although backed by the default memory manager, the kernel's virtual copy optimizations might use the page images already in memory or might fetch page images from the memory manager that backed the sender's memory. Although this can subject the receiving task to an arbitrary memory manager, it can yield significant performance improvement over the direct copying of large amounts of memory. The optimal case for a virtual copy is when the sender has used the de-allocate option. See Virtual Memory Management for additional information.

There are two options for the reception of an out-of-line or port array memory region. In option one, the default case, the received region is dynamically allocated as temporary memory in the receiver's address space, as if by vm_allocate. If the kernel transmitted a virtual copy, the received data appears in allocated space at the same alignment within a page as that of the sender. In all other cases the allocated data starts at the beginning of a page boundary. Under no circumstances does the reception of a message with this option stall the receiver by inadvertently referencing the virtual copy itself. In option two, the alternate out of line reception mode causes the out-of-line region to overwrite the bytes in a specified area as if by vm_write. If the region does not fit in the specified area, the receiver sees an error. Under various circumstances, depending on the method of data transmission and data alignments, this option might perform page manipulations or actually copy data. This operation might possibly interact with the memory managers backing any virtual copies and arbitrarily stall the receiver.

The choice of the two reception modes depends on the MACH_RCV_OVERWRITE option. When not set, all received regions are dynamically allocated. If set, the receive buffer is considered to describe a "scatter" descriptor list, of the same form as a descriptor list in a sent message. The kernel scans this list to determine what to do with each received region. If not enough descriptors are supplied to handle the sent number of regions, any additional regions are dynamically allocated. The kernel looks at each out- of-line descriptor to determine the disposition of the corresponding region. A "sending" option of virtual copy implies dynamic allocation, an option of physical copy implies overwrite with the address and size fields of the descriptor used to locate the region to be overwritten. An insufficient size causes an error return.

The receive operation is also a two-step process: de-queue the message and then make a copy in the receiver. The first step might fail because the specified receive time-out value was exceeded or because the receive was aborted (thread_abort). These situations do not affect the message that would have been received. Most failures occur during the second step causing the message to be destroyed.

There is a notification that can be requested as a result of a mach_msg call. The notification is not generated by mach_msg, but is requested by the MACH_RCV_NOTIFY option. This option causes the reply port right that is received to automatically have a dead name notification requested for it as if by mach_port_request_notification. This option is an optimization for a certain class of RPC interactions. The dead name notification on the reply port name allows the receiver of the message to be informed in a timely manner of the death of the requesting client. However, since the reply right is typically a send-once right, sending the reply destroys the right and generate a port-deleted notification instead. An optimization to cancel this notification is provided by the MACH_SEND_CANCEL option to mach_msg. Message operations are atomic with respect to the manipulation of port rights in message headers.

Section 5. Virtual Memory Management

The Microkernel's virtual memory design layers the virtual memory system into machine-dependent and machine-independent portions. The machine-dependent portion provides a simple interface for validating, invalidating, and setting the access rights for pages of virtual memory, thereby maintaining the hardware address maps. The machine independent portion provides support for logical address maps (mapping a virtual address space), memory ranges within this map, and the interface to the backing storage (memory objects) for these ranges through the external memory management interface.

The virtual memory system is designed for uniform memory access multiprocessors of a moderate number of processors. Support for architectures providing non-uniform memory access or no remote memory access is currently being investigated. High performance is a feature of the microkernel virtual memory design. Much of this results from its efficient support of large, sparse address spaces, shared memory, and virtual copy memory optimizations.

Finally, the virtual memory system allows clients to provide the backing storage for memory ranges, thereby defining the semantics that apply to such ranges.

Reference is made here to the above cited copending United States Patent Application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, entitled "METHOD AND APPARATUS FOR MANAGEMENT OF MAPPED AND UNMAPPED REGIONS OF MEMORY IN A MICROKERNEL DATA PROCESSING SYSTEM", which is incorporated herein by reference for its more detailed discussion of this topic.

Figure 6. shows the client visible virtual memory structures. There are three memory ranges, two of which have the same backing abstract memory object, but possibly differing inheritance or protection attributes. One of the memory cache/abstract memory object pairs is shown in detail with two memory object representatives, representing read and read/write access, and the memory manager task. A reserved, but un-allocated region is not shown. The region would be marked with only the reservation flag and an inheritance attribute. No other attributes apply.

Reference is made here to the above cited copending United States Patent Application by Guy G. Sotomayor, Jr., James M. Magee, and Freeman L. Rawson, III, entitled "METHOD AND APPARATUS FOR MANAGEMENT OF MAPPED AND UNMAPPED REGIONS OF MEMORY IN A MICROKERNEL DATA PROCESSING SYSTEM", which is incorporated herein by reference for its more detailed discussion of this topic.

Part B. Detailed Description of the Invention

Figure 7 shows a functional block diagram of the host multiprocessor system 100, with the IPC subsystem 122 and the capability engine 300 managing interprocess communications between two tasks 210 and 210' with threads running on two processors 110 and 112. The data processing system can be a shared memory multiprocessing system, as is shown in Figure 7, or a uniprocessor system. The microkernel 120 operating system is loaded into the memory 102 of the data processing system 100. In accordance with the invention, the microkernel includes the capability engine module 300 that manages capabilities or rights to map regions of the memory 102. The capability engine 300 is included in the IPC subsystem 122, along with several other modules, including the call scheduler 232, the message passing library 220, the temporary data module 400, the control registration module 500, the asynchronous reply module 600, the transmission control separation module 700, the shared memory support module 800, and the fast path module 900. All of these modules provide services contributing to the interprocess communications operations of the IPC 122.

The microkernel 120 forms two task containers 210 and 210', two threads 248 and 248' that belong to the respective tasks, and a data object 242. The task(A) 210 is created with a pointer 240 that maps the data object 242 into the task container 210. The port rights of the task container 210 enable it to use the pointer 240 to refer to the address space occupied by the data module and access the data contained therein. The data referenced by the pointer 240 is thus made available to the thread 248 during its execution. Thread 248 has the instructions it sequences through, processed by the processor A, 110.

Task (B), 210' also has an associated thread 248' that sends instructions to the second processor B, 112. The task (B) port rights are not sufficient to enable it to map the data in the data object 242 into its address space so that it would be available to the thread 248' for execution in the processor 112.

To illustrate the operation of the IPC 122 and the capability engine 300, the example is provided of a user program requesting that Task(B) be given sufficient port rights to enable it to map the data object 242 into its address space, thereby enabling its thread 248' to have the data available for its operations with processor 112. This is accomplished by the message 244 from task 210 to the message passing library 220 of the IPC 122, providing the port rights and the pointer 240 to the data object 242. The message passing library 220 requests the capability engine to check the port rights of the Task(B) and to check its availability and other conditions necessary to authorize the transfer of the capability to Task(B) to access data object 242. Once the capability engine has performed the appropriate comparison of port rights and other factors, it queues the call on the call scheduler 232, awaiting the availability of Task(B) to receive the message 246, authorizing the capability by providing the pointer 240 and the port rights granted by the sending task(A). The message passing library 220 can keep custody of a copy of the port rights for both the sender task (A) and the receiver task (B) to facilitate future interprocess communications between the two task containers.

Recapping, the first task container 210 is formed by the microkernel 120 in the memory 102, having a set of attributes defining a first communication port and a first set of port rights, and having a pointer 240 to a memory object 242, the first set of port rights conferring a capability on the first task container 210 to access the memory object 242. A second task container 210' is formed in the memory 102 having a set of attributes defining a second communication port and a second set of port rights.

In accordance with the invention, the capability engine 300 registers the first set of port rights for the first task container 210 and the second set of port rights for the second task container 210'. The capability engine then compares the first set of port rights and the second set of port rights to determine if the second task 210' can be allowed to gain access to the memory object 242. There is a wide range of port rights that can be attributed to a task's port space; various permission levels, security levels, priority levels, processor and other resource availability, and many others, limited only by the user's imagination. The capability engine 300 analyses these rights and can selectively enable a transfer of the pointer 240 and the first port rights from the first task container 210 to the second task container 210' to confer onto the second task container 210' the capability to access the memory object 242. In this manner, the capability engine 300 manages the interprocess communication that must take place between the many clients and servers in the Microkernel System 115, in a fast and efficient manner.

The invention applies to uniprocessors, shared memory multiprocessors, and multiple computers in a distributed processor system. Figure 8 shows a functional block diagram of two host multiprocessor systems 100 and 100' running in a distributed processing arrangement, with the IPC subsystem 122 and the capability engine 300 on each host processor managing interprocess communications between tasks with the exchange of messages between the two hosts over a communications link 250.

In Figure 8, the thread 248' of the host 100 sends instructions to be executed in the I/O adapter processor 108. The instructions sent by thread 248' for execution can include those necessary for the formulation of a message to be sent from I/O processor 108 to I/O processor 108' of the host 100'. Such a message can include the information about the port rights and the pointer 240 sent from the Task(A) 210 to Task (B) 210' with the assistance of the capability engine, as discussed above. The port rights and pointer 240 information are sent in the message over the communications link 250 to the I/O processor 108'. There, the thread 249 associated with task 211, executes in I/O processor 108' and transfers the information about the port rights and pointer 240 to the task 211. Another IPC transfer, similar to the first one described, can be carried out in the host 100' to transfer the port rights and pointer 240 originally held by Task(A) 210 in host 100, to task 211' in host 100'. The thread 249' belonging to task 211' in host 100', executes instructions in the processor 112' of host 100'. If the program executed by the thread 249' in the processor 112' includes a command to access the data in the data object 242 of host 100, the port rights and pointer 240 now mapped into or stored in the address space of task 211' enables the processor 112' to access the data object 242. The data from data object 242 can then be transferred over the communications link 250 and stored in the data object 243 in host 100', if desired. This is but one example of the role that the capability engine can play in facilitating interprocess communications either within its own memory of residence or alternately with processors having separate memories.

Section 1: Subsystem Level Interaction

The IPC 122 subsystems relationship with VM, Ports, and Tasks and Threads

1.1 The Simple IPC 122:

As with all other sub-systems in the microkernel 120, it is necessary to define the interaction/interfaces between IPC 122 and its peers. This helps in that it isolates the activities of IPC 122, easing the job of formal definition. With this kind of formal definition it also becomes possible to replace whole subsystems, providing a powerful tool for system customization. In its simplest form, IPC 122 interacts with port, task, thread, and scheduling objects. It must also work with the zone subsystem for temporary local storage. Fig. 9 is a depiction of the transfer of a simple message and the high level interaction of IPC 122 with other subsystems.

High Level Outline of Activities

```
Message→    IPC 122 Subsystem
            →ZONES: Get Temporary Storage
            →PORT SUBSYSTEM: (Capability Engine)
                 Translate Port Object (Task)
                 Push Message to Port, including transfer of port rights
                 Check Port for Receivers
                 SCHEDULER: Schedule Receiver (Thread)

RECEIVER→   IPC 122 Subsystem
            →PORT SUBSYSTEM: (Capability Engine)
                 Go to sleep on Port
                 Awake with message
            →ZONES: Free Temporary storage after copyout of message, including transfer of port rights
```

In an effort to preserve the simplicity of the IPC 122 subsystem's interface to its peers, and to provide a primitive service upon which future developers may create richer message passing systems, we have preserved the notion that IPC 122 acts only on direct data and ports as primary data types. The capability engine 300 embodies this primitive IPC 122. All interactions with ports must go through the capability engine 300. The capability engine 300 provides the calls which create capabilities on address ranges, map capabilities to virtual address spaces, queue messages, queue receiver threads, and allow explicit dequeuing of receiver threads and messages. (Though the call to queue goes through the capability engine 300 interface, queuing mechanism and policy does not belong to the capability engine 300). The capability engine 300 can also act as a synchronous or asynchronous stand alone message passing system.

In the asynchronous case, a receiver thread call will return with a message if one is queued. Or: if a receiver has been blocked waiting for a message and a message comes in from a separate thread, the capability engine 300 will unblock the receive thread and return it with the message. The capability engine 300 may act as a primitive SVC or synchronous interface if an svc call is made. In this case, the capability engine 300 checks to see if the receiving task has set up the resources necessary for the exchange. If it has, the capability engine 300 changes address space to match the owner of the target port's receive right, associates the receiver resources, and returns. If the resources are not available, the sender thread blocks. The execution entity will find itself returning from the call to the same place in the code and with the same activation environment, (in most cases this means mostly the same kernel stack) but operating possibly in a different address space.

The queuing mechanism and scheduling policies are associated with the port object and are not specific to the capability engine 300. The specific scheduling queuing policy the capability engine 300 will call may be altered on a port by port basis via calls to the capability engine 300. There are two separate sets of calls for send and receive on the capability engine 300. With one set, the content of the message field is opaque to the capability engine 300. With the other the capability engine 300 then translates all incoming capabilities contained in the message to rights in the receivers space. The second set allows the capability engine 300 to act as a primitive, asynchronous standalone message passing utility.

1.1.1 Capabilities:

From the subsystem perspective the only actual data types dealt with by the IPC 122 subsystem are direct data and port rights. This may seem surprising until the usage of port rights as capabilities is explored. Port rights may be used as a means of passing around the right to map a section of a task's address space, or an extent memory region. Hence, a message passing system could be created which passed small amounts of data directly, but created ports as a means of transferring large amounts of data via the create capability and map capability calls. The ports become tokens, or handles, to describe a portion of the task's mapped address space. The method, though workable, suffers from some measure of inefficiency as it would be necessary for a thread in a task to make a capability engine 300 call to return a capability for a specific region of its address space, pass this to the target task, and have a thread in the target task map the capability into the target task's virtual address space. Still, because of its simplicity and functionality, in short its fundamental nature, the capability provides a perfect primitive from which to build our message passing architecture.

A capability can be defined as a right to map a memory region. The region to be mapped may be part of another task's address space, in which case the capability is to be a shared memory resource, or it may be to an extent memory region. In the extent memory region case, the capability is the only handle on this memory region. Port rights may be send or send once. In this way, the originator of the shared region could either suppress unauthorized additional sharing of the region or could allow the task to which the share region has been presented to extend share region access to other address spaces through the port primitive copy_send. Share and send_once are options on the create_capability call. Though the send_once option is available on extent memory region as well as share region capabilities, it limits receiver usage without benefit. After a receiver receives and maps a capability, it is free to create a new capability on the same region of memory. The use of send instead of send_once rights is the basis for the multiple share capability and represents an alternative method for establishing binary share.

1.2 The Message Passing Library:

Simple message passing as embodied in the capability engine 300 is fully functional, but does not provide the opportunity to take advantage of many performance optimizations available in traditional message passing models. With the proper level of function provided by the capability engine 300, a coresident messaging package can be placed in the supervisor space of a target task. This library would be able to make local calls to the capability engine 300 when necessary. The space in which the IPC 122 package resides is not part of the kernel space mapped into each task, but rather is a region of space mapped through a privileged utility on a task by task basis. A task through its personality and subsequently the personality through the personality neutral services would be allowed to allocate portions of a task's address space as supervisor mode, library repositories. They would also be allowed to download trusted shared libraries into this space which would appear local to the kernel services through the system wide co-residency utility. In this way any number of customized message passing front ends might be available in a single system. The application would call the message passing library 220, not the underlying capability engine 300.

The capability transfer scheme may be powerful, but in traditional message passing it is un-necessarily expensive. The IPC 122 library (henceforth referred to as IPC 122) will continue to support the capability as the underlying logical data type but through a new data type avoid when possible the explicit creation and translation of capabilities. The new type, which is secondary in that it is strictly internal to IPC 122, will be referred to as the by-reference data type. The by-reference data type is also a secondary data type in the sense that its function can be described as combinations of capability subsystem and capability transfer calls. As an example, let us say task A wishes to send 2 pages of data to task B. Task A may call the capability engine 300 directly to get an explicit capability for the region of its address space corresponding to these two pages, pass this capability to Task B again through a primitive capability call and Task B may do a capability map call; conversely, we may send the data using our by-reference data type and a direct data size field. As will be shown, in many message passing paradigms, no explicit capability will ever be created, the capability subsystem will not be called for the passing of the data associated with Task A's address space and we will enjoy a healthy performance boost.

The by-reference data type is a pointer to a location in the user's address space and is accompanied by a direct data size variable or an implicit size dictated by the specific interface definition. (In the latter case the size would be transmitted to the kernel via the message control structure.) The treatment of by-reference regions requires IPC 122 to utilize the same sort of capability interfaces or performance driven variants that the receiver would otherwise have to call to map capabilities directly. The variants will arise out of the opportunity to skip the explicit message creation step altogether when the sender and receiver are both synchronized. These variants represent internal optimizations of the IPC 122 library which are transparent at the component level. The conditions under which synchronization is experienced and the opportunities created by it are explored later, but in general synchronization is present in all RPC cases, most send/receive IPC 122 cases, synchronous send IPC 122, and even in the general IPC 122 case whenever the receiver arrives at the message handoff point before the sender does. The juxtaposition of sender and receiver then is common and the opportunity it provides for performance enhancement makes it an important consideration.

In order to make the by-reference data type exchange roughly equivalent to transfer of a capability, it is necessary to preserve the choice of mapping enjoyed by an explicit capability mapping call. This flexibility can have a significant impact on receiver resources as a receiver may look at a capability after receiving a message and may decide not to map it. The new IPC 122 subsystem achieves this by separating the by-reference optimization on the sender's side from that of the receiver. Data sent by-reference may be received as a capability or as a mapped by-reference field. Conversely, data sent as a capability may be received as such or as a by-reference field. In order to match capability subsystem mapping functionality, it is necessary to allow the receiver to allow "allocate anywhere" and to choose a specific address at which to allocate. The IPC 122 subsystem grants this and more, it also allows the collection of by-reference regions into contiguously allocated areas (this mimics the sequential map of capabilities into contiguous regions), as well as their individual placement.

The IPC 122 subsystem must of course use capability engine 300 functions to create and manipulate memory object capabilities, but the decomposition of traditional IPC 122 function into combinations of simpler components is a significant step forward in the formalization of subsystem level interaction and architecture. Strict formalization of the proof of FUNCTIONAL equivalence between the simple message passing model and the one including by-reference data types will fall out along the following lines. 1: Transition between by-reference and capability is complete (closed in the range/domain sense) and completely independent with respect to sender and receiver. 2: Capabilities are still provided granting the full range of receive side flexibility. 3: No function exists in the mapping of by-reference data which is not available as a decomposed set of capability calls on a capability.

It should be noted that the new message passing paradigm as it is presently defined does not allow for the CONTROLLED placement of data into UNMAPPED portions of the task's address space, only MAPPED ones. UNMAPPED placement is supported through the simple model via a capability call on the target capability. There is currently no plan to include this option in the by-reference case as it can be mimicked by first mapping the region that is to be the target of the income by-reference data. As was stated above, since the passing of capabilities is supported by the message passing facility, it is not necessary from a formal model perspective to mimic every vm_map combination. The cases supported for by-reference data make sense either from a pure performance perspective or in support of a traditional RPC or IPC 122 model.

Section 2: Buffer and Address Space Resource Distribution by-reference data sub-classes

Having created the by-reference data class, the determination of location and the moment of allocation become issues of extreme importance. In the case of capabilities, the application determines the location of data by knowledge of the specific use of the target parameter in a target call. Such specific knowledge is of course unavailable to the IPC 122 subsystem. In order to communicate buffer disposition knowledge to the IPC 122 subsystem for use on by-reference parameters the subsets of parameter types must be understood and formalized. Separate data classes which are proper subsets of the by-reference data class are determined by specific restraints on location or allocation. Provisions are made with respect to the IPC 122 interface to allow receivers to communicate which of these new data classes a particular by-reference parameter belongs. These subclasses are determined by the way in which data contained in the by-reference buffers are to be used, i.e., stateless servers have no need for data associated with a call after the reply message has been returned. The temporary nature of data makes it possible to re-use the associated allocated address space, hence the existence of Server Temporary. In another example, receivers which are acting as intermediaries or proxies may not need to access a data region and therefore have no need to map such a region into their space. They may wish to opt for the capability choice. New message passing models may bring to light further fundamental class additions to this group. However, new entries should be limited to collections which are determined by a specific repeatable treatment of data either by the receiver or the send/receive pair (as in shared memory) which sets apart a group of by-reference examples. This is the basis for all current subclasses outlined below. Functionality beyond that delivered by capabilities and their explicit mapping should not be introduced into the message passing library 220. For cases where a new message passing paradigm involving performance optimizations based on roll in of additional function is required, a separate new library should be created. This library is free to borrow interface and execution path notions from the message passing library 220, but is not obligated to do so. Such a library would operate on top of the capability engine 300 and as a peer to the IPC 122 library. Message compatibility between the peers would not be assumed and any efforts to maintain compatibility would have to be made by the new library architecture.

2.1 Temporary data 400:

In RPC (remote procedure call) transfers, many send/receive pair IPC 122 (Inter Process Communication) calls and some apparently asynchronous IPC 122 calls, the receiver knows ahead of time that some of the by reference data parameters that will be received will only be needed for a short time. The time is bounded by some transaction. The transaction may be known to the system (via RPC) or be known only to the application (when it is engaged in async IPC 122 transfers). Through the IPC 122 interface, the receiver makes the IPC 122 subsystem aware of temporary nature of the data by setting a parameter level flag. The implementation of IPC 122 has placed some properties on Temporary data 400, it has been determined that it may start without regard to boundary and that it will be concatenated together with other server temporary parameters in a region of memory provided by the receiver on a per instance basis. The receiver is fully expected to reuse the buffer for subsequent calls, though the exact nature of this re-use is left to the receiver.

2.2 Permanent Data:

This is the default class for by-reference data, i.e., it is not shared, temporary, etc. Items falling into this class have no special usage constraints upon which the subsystem can base buffer disposition optimizations. As a result, without specific instructions by the receiver (please see section 3.4.3 for details), data is placed in a way that will make it most convenient for future transferral or long term residence. Further, it is necessary that default behavior be compatible with the earlier CMU based message passing semantics. These semantics included the placement of data into previously unmapped regions of the target address space. Default treatment of permanent data includes 1: starting each buffer on a page boundary. This makes subsequent removal and transferral of regions through unmapping and remapping possible. Sometimes considered a performance win, this method would be awkward if data appeared on non page boundaries and parts of other buffers shared parts of these pages. Further, the mapping and un-mapping of resources would be fraught with artifact and residue. 2: When the data region is not of page modulo, the unused portion of the last page is not used by subsequent parameters. Again this is to facilitate future mapping and unmapping. 3: Permanent data parameters are subject to the overwrite option. This provides compatibility with the earlier CMU message passing system and gives a method for call specific (or more usually server specific in the case of demultiplexing servers) disposition of individual parameters.

2.3 Shared Data

The shared data class requires specific setup and initialization by the sender and receiver. During the setup phase, the sender must explicitly make a portion of its mapped address space available to the receiver as a shared region, and the receiver must expressly accept this region as a shared one and direct it to a portion of its own address space. Thus a shared data region cannot enter a space without the explicit knowledge of the application associated with that space and conversely a region of a space cannot become shared with another task without the explicit knowledge of the local threads. Shared data support is a fairly rich package which allows one task to signal another that an arbitrary portion of a physically shared area is now available. (filled with data, cleared of data etc.) What separates the integrated shared memory support 800 of IPC 122 from specific ad hoc use of shared memory space and semaphores is the support of the paradigm in situations where the two parties do not share a common buffer. Application code can be non-local and yet still make use of the shared paradigm. Obviously the system integrator must be aware of the performance implications, but if these are found acceptable a non-local client or server is possible. Further, a formal language is established for describing the portion of space which has been made available and is known to the IPC 122 subsystem. Reflective memory techniques using special hardware can be utilized in a fashion which is transparent to the two application level parties.

2.4 Server Allocated Resources

As its name implies, this buffer sub-class is specific to RPC. In RPC the entire transaction, send and receive is described in a single message control structure. Buffer regions which will be needed for data destined for the client are, by default, brought into existence during the request (client message send). This is necessary to provide the expected semantics for an important class of procedure calls (those in which the caller provides buffer space for data it is expecting from the called procedure). For cases in which the buffer is to be provided by the server it is necessary to suppress buffer allocation by the IPC 122 subsystem. To enable the simplest possible co-residency, we will want to suppress IPC 122 level buffer allocation through the use of the server_allocated option. Even if we were willing to accept the server side always expecting a buffer and having the library routine for the local call create this buffer, there is still a performance related reason for suppression. The server may already have a copy of the data the client wishes to see. Full support of the server_allocate option means that the server is allowed to set the client sent parameter to point directly at this data, this is obviously the method of choice in a local interaction. If the server were always required to accept an incoming buffer, the local case would suffer. The intermediate library routine would be forced to allocate a buffer, and the server would have to copy data from its permanent source into this buffer. A similar scenario occurs in the remote case and, though it slows down the transaction, is less performance sensitive.

2.5 Sender (server) Deallocate

The sender deallocate buffer subclass is present in IPC 122 and on the client side of RPC. It is characterized by a wish on the part of the sender to deallocate the memory resource associated with a parameter after the associated data has been communicated to the receiver. The existence of the deallocate option allows the IPC 122 subsystem user to avoid an otherwise unnecessary call to the VM subsystem.

It can be argued that in most cases where the environment is such that the message passing is explicit the best possible performance profile will be based on buffer reuse. Buffer re-use, however, is not always practical even where message passing is explicit. Data which is mapped into a space, worked upon and then sent on its way is probably well served by IPC 122 sender deallocation.

In RPC, it is necessary to support the case where a caller is expecting the called procedure to deallocate a buffer pointed to by one of the calling parameters. Without the availability of server_dealloc, support of this behavior in the remote case would require explicit buffer deallocatian by the client side stub upon return from the send before returning to the application. RPC also supports an analogous option on the server side dubbed server_dealloc. Server_dealloc can be used on buffers associated with data the server is returning to the client, with buffers the server is receiving data on or buffers which serve both functions. In the server send case serv_dealloc behavior is the mirror of send dealloc. In the client send case, the server_dealloc function appears to operate like a server associated with a server_dealloc follows the rules of permanent buffers. This makes it easier to manipulate on subsequent calls within the server. Further, the buffer which is deallocated when the server makes its reply is the one associated with the reply data, not necessarily the one allocated on the request.

2.5 Transmission Information:

Along with the normal header fields, the message passing model provides the opportunity to gather extended information about the ongoing transfer through optional parameters. These optional parameters are mostly direct data, but where they are not, they are considered members of the temporary subclass. All direct data and by-reference pointers follow the normal header information. Data within the extended fields is mostly direct communication with the IPC 122 subsystem. Where requests are made of the subsystem, information supplied by the other party in the transaction may influence the information the kernel sends. Examples of this are NDR and STATUS. (NDR is a description of the underlying hardware level data formats, status is the functional status of the endpoint routine. Please see section 3 for more details.) Other fields such as trailer may have whole sections given over to peer to peer communication between the sender and receiver stubs.

The header, along with the direct data portion of the optional transmission control information, is sent as a by-reference parameter on the IPC 122 call. Upon return from the call, the header et. al. is returned in a contiguous fashion in the same buffer it was sent in. In the case of overrun, the data may appear in an IPC 122 subsystem allocated buffer, if the allocate on overrun option was set. Non-direct data associated with extended transmission control information is sent as server temporary. This means it will show up in the same buffer in which the header and extended direct control information appear, but may not be contiguous with it.

2.6 Memory Capabilities

Memory capabilities must be distinguished from other port rights because the IPC 122 subsystem must be able to map them if the receiver wishes. Moreover, the IPC 122 subsystem must be able to create memory capabilities from by-reference descriptions. IPC 122 must support the case of a client sending by-reference data and a receiver requesting the information be delivered as a capability. Memory capabilities may represent snapshot transferral of data or a memory buffer to be shared between the sender and receiver. Passing of a shared capability to a server does not require the server to make provisions ahead of time. The server will be able to detect the share setting of the capability and will take whatever action it deems fit with respect to mapping it.

Message Passing Outline, The Major Subcomponents

3.1 Outline of Execution Structure:

The capability engine 300 generates primitive SVC and strict asynchronous message passing interfaces. While it would be straightforward to emulate RPC, more complex IPC 122, passive servers, etc. on top of the capability engine 300's primitive message service, there would be a significant penalty to pay in performance. The IPC 122 library, rather, uses the capability engine 300 opaque message transfer options. Further, in RPC and two way IPC 122, the message passing library 220 chooses to remove the blocked receiver via the SVC call, removing the need to check for a receiver, dequeing it and then doing a thread handoff. If a receiver is not waiting, the sender is blocked and queued as it would be through the capability blocked sender queue call, no explicit message is created. This is very important in that it allows most by-reference transfer to occur without the creation of an explicit capability. Such a data transfer can occur without any consultation with the capability engine 300. Two sets of diagrams will serve to illustrate synchronous and asynchronous message transfer via the message passing library 220 and capability engine 300. When the receive thread is the first to arrive at the message port asynchronous handling behaves as the synchronous case. In the synchronous case, the execution path for the reply is not outlined explicitly. This is because it is an instance of example 2 of the async case. (send with receive waiting)

Example 1 in Fig. 10 outlines the path when a message is sent to a port upon which there are no waiting receivers. In the case of an RPC the application calls a library stub (not shown) the stub emulates a local procedure call for its caller. The stub assembles the pieces of a call specific message and traps to the message passing library 220. The message passing library 220, which exists in the supervisor mode, makes a local call to the capability engine 300, an SVC. In our example above, no one is waiting to receive the message, the message passing library 220 sets up a continuation (if desired) and sends it on the svc call. There is also an option on the call to use the continuation only for thread handoff and not on thread block. The capability engine 300 based on these options, blocks the sender with or without continuation. The capability engine 300 again calls a customizable queuing procedure.

In Example 2 of Fig. 11 we again have an incoming send message but this time the capability engine 300's check for waiting servers meets with success. The proper server is targeted for thread handoff. The capability engine 300 now has both the sender and receiver and can proceed with thread handoff. After returning from thread handoff the message passing library 220 can proceed to transfer the message directly. It should be noted that at no time has the message format or content been exposed to the capability engine 300. This gives the message passing library 220 full flexibility in choosing message format AND content. Future libraries could move any sort of buffer allocation of data transformation schemes deemed useful into this model. The message passing library 220 proceeds to move direct and by-reference data directly between sender and receiver. The only kinds of data which will require calls to the capability engine 300 are port and capability transformations. This includes the direct transfer of capabilities and ports but also the mapping of a capability (the server requests an income capability be mapped) or the unmapping of one. (the server request an incoming by-reference buffer be received as a capability) Scheduling again takes place without the intervention of the capability engine 300. The sender or client is already blocked and does not require another call to the capability engine 300 unless the client is to wait upon an explicit reply port. (If the server is not accepting anonymous reply, not guaranteeing the reply will be returned by the entity now receiving the message.) The scheduler is called if the server is to run with the receiver's scheduling properties.

In example three, of Fig. 12, we see things from the receive side. A receiver arrives, only to find there are no waiting senders. The receiver blocks through the Capability engine 300. The library of course decides upon the disposition of the block, i.e., whether or not to block on a continuation. It is the library's responsibility to guarantee against the arrival of a send while the receiver is in the process of blocking, or to check once more for senders after the block has occurred.

Example 4 of Figure 13 is identical to example 2 except it is from the receiver's vantage point. At user level (non-supervisor mode) a message is either cobbled together directly (IPC 122 send/receive) or created in a server loop designed to assist the target end point of an RPC (emulate a local call to the callee in a procedure call transaction). In either case a trap is made to the supervisor level message passing library 220. After making the call to the capability engine 300 to get a sender succeeds, the message passing library 220 finds itself with a sender and a receiver. It proceeds to transfer the message as in example 2 but does not need to execute a thread handoff. The scheduler is called if the thread is meant to run with the client's scheduling properties. The client will block again. As in example 2, the capability engine 300 will only be called if the reply port is explicit.

As with the synchronous case the message passing library 220 begins by checking for a receiver, when one is not found however, the asynchronous nature of the interface in Fig. 14 requires the library to engage in the expensive business of creating a formal message. Direct data is copied without transformation, port rights are generated according to their dispositions and placed in the message, capabilities are created for all by-reference parameters and capability rights are pushed according to their dispositions. All of these activities require the direct support of the capability engine 300. The message is queued through a call to the capability engine 300.

In this case of Fig. 15, the asynchronous model behaves similarly to the synchronous one. Since it is not necessary to create an explicit message, the performance of the combination of examples 2 and 3 of the asynchronous case will be significantly better than that represented by the combination of asynchronous examples 1 and 4. The only difference between the asynchronous and synchronous example 2 cases is the lack of a pending reply in the async case. Without the need to wait on a reply, the sender is free to return.

Example 3 of Fig. 16 is identical to example 3 in the synchronous case. Experience of example 3 behavior determines that the sender will experience example 2 behavior upon send. This means that performance conscious users of the asynchronous model should try to have the receiver arrive at the port first whenever possible.

Example 4 of Fig. 17 calls the port specific send queue and recovers the appropriate message based on queue function specific criteria. The message is made up of direct data which can be transferred to receiver without transformation. It may also contain ports and capabilities. The ports and capabilities must be translated via calls to the capability engine 300. Small by-reference fields may be masqueraded by the message passing library 220 to avoid the overhead of capability creation at the time of explicit message creation. The receiver may still choose to receive the field as a capability, but if it is receive as a by-reference buffer, the capability engine 300 will not have to be called.

3.2 Message Structure:

The message structure associated with the message passing library 220 contains the functional elements that one would expect in any message passing interface. It of course has provision for the buffer disposition options outlined above as well as the primitive data types. However, the message structure differs from many other systems in that the fields associated with overall transmission control, those residing in the header, do not have to be contiguous with the rest of the message structure. The message structure identifies 4 separable entities. The header points to 2 of these and the message parameters themselves point to the last. The four entities are 1: The Header 2: The message control structure (contains information about the specific parameters associated with a specific call) 3: The message (the direct data, by-reference pointers, ports, and explicit capabilities associated with a call) 4: The by-reference regions.

There is no restriction on allowing the regions to be contiguous. It is possible that there is some small performance advantage in having them all in a contiguous form but if they are formed at different times, it is not necessary to recopy them just to make them continuous. Fig. 18 is a diagram outlining message layout.

The message parameter information individually describes each parameter in the message buffer. The description fully defines the parameter data type, size (either directly as a fixed size or indirectly through a pointer to a count parameter.) and disposition of the buffer associated with the data in the case of by-reference.

Throughout the design phase of the message passing library 220, one must be mindful that the performance exhibited by an implementation can be greatly influenced by the layout of associated data structures. This has influenced the layout of the bits of the parameter descriptors of the message control structure as well as the separation of control, message, and transmission information at a grosser level. It was further realized that in many of the important modes of use, the information associated with the substructures was generated at different times and in different places. It was only good programming form to reflect this in the overall design of the message structure. Upon further analysis this nod to good form turned out to yield significant performance gain.

3.2.1 The Separation of target endpoint message data

In an application environment where the endpoint application is aware of, and directly participating in, message passing the separation of message data is of limited use. If the data to be passed is large, the wide variety of by-reference data types is available to avoid excessive byte copy. However, there is a large and interesting set of uses of message passing in which the endpoint application calls a proxy library routine to do the actual message passing trap. These uses include the entire domain of RPC and some interesting IPC 122 cases. If the endpoint application makes a copy of a rather large parameter in the act of a calling a proxy user level message passing library 220 routine, it would be nice to be able to use that copy instead of recopying the data just to make it contiguous with the message header. Again, if the message is small or the system stack conventions are not know to the proxy service the proxy can always fall back to message data copy.

By knowing the address of the parameters sent to the proxy and assuming the parameters are contiguous and in a known order, the proxy may pass the address of the parameter block as the address of the message head as shown in Fig. 19. In support of this, an extra type was added to the by-reference sub-types outlined earlier, the pointer to a by-reference parameter. This is very popular as a means of returning an altered pointer to a data structure in languages such as "C" which view the direct altering of an argument by a called function within the scope of its caller to be a violation of scope rules.

3.2.2 The Separation of Static Message Control Information:

Of the pieces of the message which have been isolated into sub-structures, among the most important one is the message control structure. Performance gains realized by the separation of the message control information promise to be significant across the whole spectrum of supported message passing. It should be emphasized that the information contained in the message control structure completely defines the transaction from the endpoint's view. (Receive side override which can be viewed as an exception to this will be discussed in section 3.2.4.3) The information pertaining to message invocation contained in the header is a dialogue between the caller of the message service and the message passing library 220. Though information can be passed between the two endpoints via the header and its optional transmission control fields, setting an option on the send side does not require the setting of an option on the receive side and visa versa. Fig. 20 is a high level sketch of the message control structure.

Primary descriptors have a one to one, mapped and onto relationship with the parameters of the message, the first corresponding to the first parameter, the second to the second and so on. Further, the primary descriptors are required to be contiguous. In this way it is possible to find the descriptor corresponding to the 3rd parameter by simple offset from the beginning of the descriptor section of the message control structure. Primary descriptors may not be large enough to carry all the state information necessary for a particular parameter. If this is the case, a field of the primary descriptor points to an offset within the message control structure which corresponds to the start of a secondary descriptor. Secondary descriptors come in different sizes, appear after the primary descriptors and in no particular order.

Full definition of the message format within a compressed and carefully thought out structure is significant from an interface definition perspective. Server's can check the match between the message format they expect and the one the sender has provided with a simple binary comparison check of the message control structure sent by the client. When a match is found, the server will be guaranteed that pointers within the message will be pointers, ports will be ports etc. It does not guarantee semantic meaning to the associated data of course, but it does mean the server is protected against random pointers and random values for port rights. The server is guaranteed of this because the message control structure (and server provided overrides) hold the sole determination criteria for the message parameter format.

What further makes the message control structure special is that it is defined prior to execution. To avoid un-necessary work, the proxy may point the header based message control structure pointer to a fixed copy of the entity in BSS or other storage, thus avoiding the need to create the structure in temporary local storage each time the proxy function is invoked. This prior definition of the message control structure is important for another reason. Conventions can be set up which allow the transfer of messages based on pre-screened message control structures. The sender need only supply a registration label. This will avoid not only the sending of message control information but also the runtime comparison between sender and receiver for non trusted sends.

The separation of the Message Control Structure is helpful in another way, makes it easier to leave it off. If a message only carries direct data, there is no need for message passing level translation or intervention of any kind. Such messages have been dubbed "SIMPLE" messages. Simple messages can be either one way or two way, RPC or IPC model. A simple message may still require a message control structure if the server wishes to test it for compatible format. This should be a very limited case however. If the server is expecting a particular message or recognizes a group of message id's, a simple message of the wrong format behaves no differently than one which simply contains garbage data. The only case where a server might need a message control structure is on messages containing variable simple data format not distinguished by message id. Unless the data is self-defining, the receiver would have to look at the message control structure to find the parameter boundaries. In the case of simple messages, the sender is not required to supply a message control structure to the message passing library 220 and the message passing library 220 does not supply one to the receiver. For transfers which the receiver needs the message control structure, the simple option should be turned off.

The message control structure has been set up to define EVERY parameter being sent. This is important for receivers which accept messages which are not pre-defined. Without the definition of every parameter, the server would not be able to parse the incoming message. There have been efforts to improve the performance of message passing by declaring all direct data to be a single field. Experimentation with prototype code on the new message control structure has shown that parsing through direct data fields has almost no performance impact. (The loop to parse direct data consists of checking a bit in the parameter disposition field, and upon realizing that it is direct data, adding a count field value to the offset within the message data structure to point at the next parameter. The bump to the next parameter descriptor and the loop check are the only additional actions.) Even with this evidence there are some who might still argue that the overhead is unnecessary. In the unlikely event that some message might benefit from the coalescing of direct data, such coalescing can be done at the proxy library level. The proxy can re-arrange the message fields, putting all the direct data fields together and labeling them as one field for the transfer. In this way the message passing library 220 can preserve within its model the convenience of identifying all parameters without any compromise of performance, real or perceived.

3.2.3 The Separation of Transmission Control Information 700:

The transmission control information subsection of the message structure consists of the header and an optional group of transmission variables. The information contained in this subsection is characterized by two things. First, it is information that is presented directly by the caller of the message passing library 220. Second, whatever its origin or final use, the fields and most often the data are parsed and interpreted by the caller of the message passing library 220 and the library itself. The motivation for the separation of message control information from the rest of the message is the same as that for the separation of message data and static control information. The collection of fields found in the transmission portion of the message is created and manipulated at the same time and by the same routine regardless of the message passing model. (IPC 122, RPC by-proxy) This guarantees that there will not be unnecessary copying. Further, the strictly enforced point of interaction for the transmission section is the message passing library 220. This preserves the very important role of the message control section as the one place to look for the sender/receiver dialogue determining message format, not just message buffer format. A sender can only influence the format of a message delivered to a receiver through the message control structure. The format of the message buffer is completely determined by the message control structure and the overwrite buffer (The overwrite buffer allows the receiver to exercise local override on final disposition of capabilities and by-reference regions. Please see section 3.2.4.3 for details). The format of the header returned from a call to the message passing library 220 is determined by the options chosen in the transmission control section at the time of the call. Hence, a receiver will be returned a message whose header format reflects the receiver's transmission control section requests at the time the receive call was made. The message buffer format on the other hand, will reflect the data sent by the sender and whatever the sender's message control structure dictated expecting where by-reference and capability disposition has been influenced by server use of the overwrite buffer.

If the caller wishes to know or influence some specific aspect of the message transfer, it interacts with the library through the transmission control section. Data can be passed from sender to receiver through the transmission section but this data is interpreted by the message passing library 220. In the case of sender to receiver communication, the interfaces are defined such that there is always a default behavior which is acceptable from a format perspective regardless of the remote party's actions. In this way the remote party's choice of transmission control options has no influence on the format of the local message. This is critical in maintaining the message control structure as the only source for message format determination where a sender may influence the format of a message received. STATUS, Trailer_Request, and Message_Control_Structure request.

A simple example of direct interaction between the message passing library 220 and a receiver can be shown with NDR request. When a message is sent the sender has the option of including an NDR_Supply parameter. This is only done if the primitive data types upon which the message data is based do not match the host machine. If the NDR_Request option is active when the message is delivered, the message passing library 220 will by default pass the NDR information of the host machine. If the sender opted for NDR_Supply then the message passing library 220 will pass the information offered by the sender.

Another important capability of the transmission control system is its ability to pass uninterpreted data between a sender and a receiver. Such data can be passed from proxy to proxy without altering the endpoint message buffer via the trailer. Certain fixed fields are present in the trailer including sequence number and security token, beyond this is an open data field. If the size of the trailer is fixed by prior agreement, and the sender sends data, and the receiver requests it, the trailer may arrive in the direct data optional header area. If the amount of data in the trailer varies from call to call, the receiver may wish to request the by-reference version of trailer_request. (The direct data version of trailer_request includes a count parameter, the count sent by the receiver is the maximum that will be received; more incoming data is truncated. This count variable is changed by the message passing library 220 to reflect the amount of data sent back up to the maximum. To receive any size up to the space provided in the temporary data 400 buffer the receiver must use the by-reference version.) In either case should the sender not provide a trailer, the trailer received will only contain the defined fields requested. If none are requested, the size could be zero. The area beyond the defined trailer fields is passed by the message passing library 220 to the receiver just as it was sent. The method the receiver decides to obtain the trailer information by has no effect on the sender. The sender is free to send the information either directly or by-reference.

When a caller of the message passing library 220 prepares a call, it sets up a header structure. This header structure sits in a buffer which must be large enough to accept, not just returned header information but also direct data associated with the transmission options the caller has requested. This also includes room for any direct data (by-reference objects) requested. This implies that by-reference regions associated with transmission control parameters are considered server temporary. As will be detailed later, when a server in an RPC or the target of a 2 way IPC 122 calls the message passing library 220, the header sits in a buffer which must not only be prepared to accept all of the transmission control information as outlined above, but also the server temporary data 400. The format of the returned buffer is, header at the top followed by direct optional control information, followed by server temporary fields, including those associated with the transmission control information.

Figure 21 is a diagram outlining the transmission control structure. The fixed portion of the header determines the kind of message passing, i.e., Send, Receive, Send/Receive, RPC, IPC 122, the kind of reply port in 2-way messages. The optional portion of the header is determined by the optional transmission flags field. Each of the optional fields corresponds to a bit. When present these fields must appear in sequence. For optional transmission fields which are by-reference. A sub-field of the optional field entry for that parameter will be used to point into the temporary buffer address area. Another sub-field will describe the size of the indirect buffer in bytes.

3.2.4 The Relationship Between Sender and Receiver Control Information:

All message passing systems must deal with the problem of coordinating sender and receiver message format and identification. Some punt, assuming that the question of message format is settled by the sender and receiver outside of the message passing paradigm. Others pass partially or fully defined messages that the receiver must parse to determine what it is and whether or not it should be accepted. Both points of view have their advantages. In an embedded system sending fully trusted messages, it is hardly necessary to burden the processor with generic message parsing. On the other hand, in the general message passing operating system, there is a real need for non-trusted communication between senders and receivers where the receiver must verify message format. General message passing also makes use of generic receive servers which parse a message to determine its format. With the separation of message control information, the message passing library 220 can efficiently support both paradigms.

Except in the case of simple messages, the sender must provide a message control structure when a send message call is made to the message passing library 220. This convention is absolutely necessary in the case of asynchronous messages where server input simply may not be available. Although not absolutely necessary in the synchronous cases it does provide a discipline. By requiring the supply of a message control structure from the sender, the receiver always has the option of checking the incoming message format. Further, the number of nonsense messages delivered from non-trusted clients is likely to be lower. If the client sent a message and relied on a server message control structure to parse it, some percentage of the time an incorrect message would get through based on the ability to incorrectly but undetectably interpret the client message parameters. A non-trusted client would then be sending garbage data to a server. If the client is required to send a message control structure, the server checks the non-trusted client message control structure, avoiding the receipt of garbage data. (The client can always deliberately send garbage data, of course.) Having the sender supply a message control structure also reduces the possibility of unintentional damage to the client. If the client were to send a message to the wrong port in the server message control information paradigm and that message were to unintentionally succeed the client might lose large tracks of data to unmapping and overwrite, i.e., a client may send a message to a server, expecting that there are two direct parameters. The server believes the first parameter is a by-reference and that further, the associated buffer is to be removed after the client send. Now if the data in the client send just happens to look like a valid address, the client will unintentionally unmap a portion of its address space.

Fig. 22 shows 2 examples of message control structure usage with out accepted convention, client supplied message control information.

The message passing library 220 consults the sender supplied message control structure to translate all non-direct data parameters. The server, however, is expecting messages of only one format, or in the case of a demultiplexing server, messages whose format is determined by the message id. The server, therefore, does not request the message control structure and acts on its assumptions. Such a server could be damaged by a client either intentionally or unintentionally sending a message of the wrong format.

With the receipt of the Client's message control structure the server is now free to check the format of the incoming message against expectations. If the server is demultiplexing, the message id is checked first to determine which amongst a set of message control structures this particular incoming entity should match. If the format is unknown, the message control structure is consulted in order to parse the message data as shown in Fig. 23. This last scenario is most likely when the server is acting as an intermediary for another server. The use of the message passing interface to implement a communications server can make a good example of the power of the message passing library 220. For two communicating nodes the integrated shared memory data types can be used. If the nodes share common memory (or hardware supported mirrored memory) the transfer can take place without overt memory copy. If not, the transfer of data occurs automatically without adjustment to the communication code.

3.4.1 A Fully Defined Send-Receive Compatibility Check

Even if a server and client have fixed on a message format, or in the demultiplexed server case, a series of message id pared formats. The server may not trust the client to do the right thing and send the appropriate message. Actually verifying the message format has historically been a dodgy affair. Changes to interface or missed subtleties often left holes in the check. Further, the more complete the check the more costly. The architecture of the message passing message all but eradicates these difficulties. All the information required to describe the data types found in a message buffer can be found in a message control structure. Further, the portion of a message control structure associated with the definition of incoming parameters contains no other information. This makes it possible to do binary comparisons of server stored message control templates with the incoming client message control structure. The distillation of message buffer information is such that the average parameter is fully described in 8 bytes. Thus the layout of the message buffer for a 4 parameter interface might be checked by a byte to byte comparison of 32 bytes. The fact that other portions of the interface, like those associated with transmission control protocol are described elsewhere means that there will not be an unnecessary restriction on transmission options in a byte compare protocol check.

The RPC system should be noted here because the message control structure describes buffers and buffer disposition for both the request and reply. It is very reasonable that a server would want to support clients that chose different local buffer disposition options. As an example let us consider 2 clients which both want to interact with a common server. They both want to send a by-reference field to the server. One wants the buffer removed after the send, the other wishes to retain it. It would be awkward if the server were to reject one of these two clients just because neither of them was trusted. The bits of the parameter disposition have been set so that this case can be handled. There is a field of bits associated with client buffer disposition. (Bits 23-18 of the flags word.) By applying a mask to these bits in the template and the client derived message control structure before the binary check, the server can service both clients in non-trusted mode.

The example above brings out one other important point. The check of send/receive compatibility is not only optional, it is user level. Though the user level libmk library support will include the binary byte by byte check and the client option mask override for RPC message control structures as callable macros, the server is free to fashion any sort of partial check it sees fit. For example, allowing clients which send a buffer as temporary as well as those which send it as permanent with the dealloc flag set. (See sections 2.1 and 2.4 for a description of the data types.)

3.2.4.2 Control Information Registration 500:

The distillation of message control information and the existence of simple messages which do not require control structures, the flexibility of the send side check and the option to leave it out, all have significant functional and performance implications. However, there is one more opportunity for performance optimization which gives non-trusted clients almost equal performance to the non-checking trusted case. Further, it speeds up both trusted and non-trusted by avoiding copying the message control structure into message passing library 220 space on a call by call basis, even on complex messages. The method involves message control structure registration 500.

A server wishing to participate in registration, makes a registration call for the message control structures associated with the server's set of interfaces. The registration call parameters are the message control structure, the associated port, and a placeholder for the returned registration id. The message control structure becomes registered with the port for the duration of the life of that port. In this way senders who acquire that registration id will be guaranteed that it is valid for the life of the port. A client wishing to send messages via the registration service, contacts the server with a simple call, sending the message control structure, possibly containing a message id, and asking for the associated registration number. The server is free to run what checks it likes, but in practice absolute compatibility is required. Should the server detect for instance a difference in client local buffer disposition and pass back the registration id anyway, the client would be damaged upon the use of that registration id. The server may fail a registration request which does not match exactly or register an additional message control structure for that particular message id. The server would then be responsible for checking both registration numbers for that particular message id, the server template registration number and the client registered on. The server should also keep a copy of the client message control structure on hand to check against future registration requests. If a client is refused a registration number, it is still free to attempt non-registered transfer.

The registration of message control structures for servers which persist over long periods is certainly indicated for both trusted and non-trusted client-server pairs. It will be most significant in the non-trusted case, however, since it removes the need to copy the message control structure to the server and do the call by call check for format compatibility. A registered server will work with both registered and non registered senders. Therefore, if a sender is only going to interact with a receiver once or twice it may not be deemed worthwhile to do an extra call to retrieve the message control structure registration id.

Fig. 24 shows a diagrammatic depiction of a message registration and use by a sender. When the client attempts a send with the newly acquired registration number, the message passing library 220 checks a port associated queue for the proper message control structure. The message control structure is local to the message passing library 220 and thus a copying of the control structure is avoided. Further, on RPC's, it is necessary to keep the message control structure handy while the client is awaiting replay, one control structure is kept for each ongoing transfer. In the registration case, only a registration number need be stored. The message passing library 220 is set up such that the client must request the registration information of the server for two important reasons. First, it reduces the code which must be maintained in the message passing library 220. Second, the server maintains full flexibility in determining who matches registered message formats and who does not. Use of the overwrite option and reply overwrite can make a wide variety of incoming message formats compatible. It is up to the individual server to sort through this and support the set of formats it sees fit.

3.2.4.3 The Overwrite Buffer

Receivers who wish to influence the placement of permanent data and receipt of capabilities in their space upon the acquisition of a message must supply an overwrite buffer. The types of data influenced by an overwrite buffer are 1: Permanent data (note: the permanent, by-reference choices also include the server dealloc cases.) and 2: capabilities. It is possible via the overwrite buffer to request that a capability be written into a mapped area of memory. Or have an incoming permanent by-reference buffer transformed into a capability.

Overwrite Buffers are supplied via an option in the transmission control structure. As such of course, they only affect the local caller. Overwrite has additional functionality. The overwrite buffer considers capabilities and by-reference permanent regions to be enumerated or indexed as encountered. As the incoming message is scanned, the first encountered capability or permanent by-reference region is influenced by the first descriptor in the receive overwrite buffer, the second encountered, by the second descriptor and so on. Intervening parameters of other types have no effect. The only exception to this is when the receiver chooses the gather option. In this case data from multiply by-reference regions or that associated with capabilities is concatenated together and written into memory starting at the location specified by the overwrite descriptor. Any number of descriptors may be concatenated in this way and there is an option to make the number strict or "upto". In the strict case, exactly the stated number of regions must be found to fill the gather descriptor area or an error is returned. In the "upto" case, if the number of regions specified in the descriptor is larger than the available number of regions in the incoming message, the message proceeds anyway. Descriptors in the overwrite region which account for regions than are found in the message are ignored. Likewise, if the overwrite descriptors account for fewer permanent by-reference and capability parameters than occur in the message. The parameters beyond those enumerated by the overwrite structure behave as if the overwrite option had not been exercised.

Use of gather often necessitates the request of send message control information by the server so that the actual size and number of permanent regions and capabilities will be know. The control structure must also be consulted to find the direct count fields associated with dynamic by-reference regions.

In the case of RPC it is necessary for the server to construct a message buffer for the reply which in the format the client is expecting. In two way IPC 122 of course this is always true as there is no protocol link between the format of the send and the receive. Fig. 25 is an example of overwrite use.

3.2.4.3 Reply Overwrite Control Information:

When a server redirects the placement of data on a by-reference region using the overwrite option, care must be taken to ensure the post, or reply processing is appropriate. An RPC style interface might well have been set up to deallocate a by-reference region using the server-dealloc option. If the server has re-directed by-reference data to a region that it wishes to persist past reply delivery, it must pass back an altered message control structure. Upon detection of the server side reply side control structure, the message passing library 220 scans it for server side buffer disposition overrides. The message the client is expecting back in the case of RPC is of course in the client format. It is up to the server to put together the appropriate message buffer. It might have been possible to send null buffers back on fields upon which the server-dealloc option was set for buffers which were only passing information to the server. This however, was an insufficient answer for the buffers being used to send data both ways or just to the client.

Section 4: Message Passing Models

The capability engine 300 is defined to create a simple and general message passing service. It has done so by formalizing all transfers as either direct data or capabilities. In a port based message passing system such as MACH, such ports can be used to pass access to any transformation. The message passing library 220 carries with it the function of the capability engine 300 but it has sought to raise the performance level by taking the mapping transformations which must be done explicitly in the capability engine 300 paradigm and creating a language to express there transfer without creating formal capabilities. If the two endpoints of a transfer both use the message passing library 220 then a mapped area of a sender's space can be described in a message and the place to write it in or map it in can be described for the receiver.

Asynchronous messages still require the creation of capabilities because the data sent must be captured before the sender returns and the receiver is either not yet known or not yet ready to receive the data. Synchronous interfaces on the other hand need never create intermediate capabilities for the by-reference data types, because the sender must pause for a pending reply anyway, the point of synchronization for the client is not the return from the send but the return from the reply. This allows the message passing library 220 to pause the client before message creation and to proceed only when a receiver is available and the transfer can proceed from task space to task space without an intermediate message. It is clear then that the message passing library 220 must also formalize the type of transfer (asynchronous vs synchronous).

It can be further recognized that there are really two kinds of synchronous transfer, one in which the semantic meaning of the reply is directly tied to the send and one in which the two are disjoint. The message passing library 220 was designed to support a wide variety of message passing models. Its base function for data transferal is the same as that for the capability engine 300. But it also includes tools to facilitate non-layered support of popular forms of Remote Procedure Call and Interprocess Communication.

4.1 Remote Procedure Call

Remote Procedure Call or RPC can really be distinguished from the message passing library 220 function, larger cloth from which it is cut, by a series of restrictions. Some of these are listed below.

1: A send does not return from the call to the message passing library 220 until the message has been delivered to its target.

2: The data in the send and receive portion of an RPC is semantically linked to the extent that a message of the same format is sent and received.

a) the incoming and outgoing message share the same format

3: The RPC system must be capable of acting as a proxy. It must be able to simulate the call of a local procedure by acting in place of that procedure. It must transfer the associated data to the task space where the remote procedure lies, await the remote procedures processing, return the results to the callers space and finally make all the incidental changes like buffer removal or creation for the class of procedure calls supported.

The third point may not seem like a restriction, indeed the proxy notion accounts for the separation of transmission information as a substructure of the message. In a sense though it is. In the proxy case, the parameters in a message buffer are exactly those sent by the initial caller. In many languages this creates some restrictions. In C for instance, all the direct variables are of fixed length and data cannot be passed back in a direct variable. This allows the RPC subsystem to make performance enhancements. Opportunities for performance enhancement based on specific use is in fact the reason for formalizing the support of RPC, the restrictions which distinguish it allow for additional optimizations.

Restriction 2 is actually a guarantee by the message passing system that a client initiating a call will not succeed in starting that call and activating the associated server in a non-restartable way only to find out that a loosely paired reply does not match the client's expectations. The semantic link between request and reply has implications for the message control structure and the registration service. Because the request message and the reply message must have the same format, it is most natural to have the message control structure contain information for both the send and receive, coalescing the control information rather than sending two structures. Coalesced or not, the fact that client must declare the entire operation upon the initiation of the RPC has an impact on message checking. A server utilizing the overwrite option may accept a wider range of incoming client messages and may have to adjust its message control structure check because the clients send different remote buffer disposition information. Restriction 2 also accounts for some additional options in registration. The need to accept and register possibly more than one message control structure format grows out of the asymmetric nature of the client/server relationship. The server registers the client's message control structure. If there are two clients which send exactly the same format message but wish to receive the reply data differently, the server must register two message control structures to support them both.

The implications of restriction 1 have been considered in detail in section 3. The adoption of synchronous message passing not only leads to lower CPU overhead in data and resource transfer, it also decreases kernel level resource utilization and makes it more predictable.

Fig. 26 is a diagram of the RPC transfer. The message control structure is kept in the message passing library 220 while the server is active in anticipation of the reply. If the message was complex but was accompanied by a large amount of direct data, the server can avoid sending this data back on the reply by sending an override message control structure with zero size direct data fields. The message passing library 220 will use the override message control structure to find the by-reference, capability, and other port fields in the message buffer sent from the server and will fill in client buffers, or update the client's double indirect pointers as appropriate. The client message buffer is, of course, not written back to the client.

4.1.1 Scheduling Alternatives and Threading Models:

There are two major models of RPC support with respect to scheduling. The active and passive server models. In the active case the scheduling information associated with the client's request is that of the server thread. In the passive, it is that of the client. In the active model, the server can be observed to directly commit a thread to the receipt of a message on the target port. The client then sends a message to this port and blocks waiting for the reply. The server thread returns to non-supervisor mode with the message and proceeds to process it, returning with a reply when processing is complete. In the passive model, the server as owner of a port, prepares a thread body, (it prepares state and a set of resources for an incoming kernel level thread). The client does not so much send a message as enter the target server's space with the kind of restrictions associated with a traditional kernel level service call, i.e., start execution at a target mandated point, process incoming parameters along previously defined lines.

In the case of RPC the assurance that the client will block while the server is working on its behalf is very helpful in supporting elements of the passive model without having to expose an actual passive or thread migrating model to the user level. First, all kernel level temporary resources associated with a client thread at kernel level may be borrowed by the server. The thread stack and other temporary zone space are good examples. The client prepares a message for transfer, the server is then allowed to borrow the buffers which hold the results of that preparation. In this way, there is no distinguishable performance difference between the two models. Indeed, whether or not a transfer is recognizable as thread migration has more to do with the naming of kernel level resources than with the actual implementation at kernel level. As an example, in a recent paper, calls to threads were transformed to associate thread level markers such as the thread port with the thread body instead of the thread proper. In this way, the portion of the thread associated with migration, the thread shuttle will become effectively anonymous. The effect could have been achieved another way. Staying within the paradigm of active threads one can enumerate the characteristics of thread migration as separate options. The most important is, of course, scheduling. If the server thread in the active case inherits the client's scheduling characteristics, and the kernel elements of the thread are anonymous, there is near performance and functional equivalence between the passive and active models.

In the active model an actual runnable thread is created on the server side. This may or may not be used for other activities, in either case it is eventually put to sleep awaiting a receive. If the port is a passive RPC port, kernel level resource, even the schedulable entity may be discarded. (A port level scheduling information template for lost state would have to be made available for aborts.) When a client arrives at the port with a message, the client loans its kernel temporary resources and its schedulable entity, effectively its shuttle to the server thread, now effectively a thread body. The client entity, now effectively a thread body is blocked or put to sleep on the reply port.

There are some advantages in exposing a passive thread model to the user level. Certainly, easier thread body resource management is one of them. If an active model user wishes to move the "thread_body" from one waiting port to another, it must exercise an abort. Exposing the actual resource queues for thread bodies to the application level would allow the user to move resources with simple pointer manipulation. Further, creation and destruction of thread bodies is less expensive in the exposed case. This might give a small advantage to the extremely dynamic server case. Depending on the exact nature of the interface it is also possible in the exposed case to allow for resource pooling between receive ports. Letting separate ports draw upon common thread body resources. The method would be somewhat more flexible than port_sets in that thread resource could be subsetted but most of the equivalent pooling capability could be supported through port sets. It is thus possible to support the functionality and performance of the passive model (when that model uses anonymous kernel resources) with an active interface. The world of asynchronous messages, will be somewhat more difficult. It is all but impossible to maintain equivalence between the active and passive models when it comes to one-way sends.

User level models demanding the availability of state information on the depth of calls and path of migrating threads would, of course, force the exposure of a migrating thread model. Shuttles would no longer be anonymous and would carry with them information regarding the recursive depth and path of ongoing calls. Only such a direct requirement for message passing library 220 supported state is expected to force the need for migrating thread exposure however. Even reasonable abort semantics appear to be supportable without such direct exposure. See section 4.1.9

4.1.2 Client/Server Juxtaposition:

Client/Server juxtaposition is characterized by the synchronization of the client send and server receive. In the case of RPC, if the server arrives at the receive port before there are any messages to receive, it blocks. If the client arrives before the receiver it blocks until the receiver arrives. This in effect guarantees simultaneous access to both the client and server space for the purpose of message transferral. Though client/server juxtaposition can be achieved in some circumstances in asynchronous communications it cannot always be guaranteed as it is in the case of RPC. If an asynchronous send is attempted on a port upon which there is not a waiting receiver, the message passing library 220 must produce a message and allow the sender to continue.

In synchronous transactions the ability to guarantee that a sender cannot continue until the receiver has obtained the data snapshot means that actual messages need never be created. This minimizes expensive capability translations, message creation, message parsing and free operations. When a message is created, all by-reference types essentially revert to capabilities. The memory regions associated with the by-reference area must be copied (in one form or another, copy on write, copy maps and such are beyond the scope of this paper) and pointed to out of the message. This effectively creates a capability. The capability is anonymous which saves the target space mapping costs but it is still quite expensive.

Even in the case where an explicit message must be created, by-reference types are superior in performance to capabilities because they still allow the receiver to map or write incoming data without doing a specific call. Further, some small by-reference fields might avoid capability translation by temporary conversion to direct data. This seems especially likely for the server_temporary examples.

Assuring client/server synchronization also reduces the need for kernel level resources and leaves the remaining resource needs more predictable. In an asynchronous world, system lockup through resource over-utilization can occur when too many messages are left waiting in queues. An example can be easily constructed. Thread A sends a message to thread B. B, however, is busy processing an earlier request (possibly from A). To process this request, B must post messages to several other tasks. Each of these messages requires a large amount of space. Each of the subsequent tasks must, in turn, post a message. The system designer made sure that there would be enough resource to run the request, but failed to take into account the storage that additional waiting requests on the thread B would use. The system halts, or fails, unable to create the messages that the tertiary threads need created in order to service thread B. This particular problem can be overcome, and indeed memory limits and such can be placed on ports in an effort to manage the problem. Nevertheless, it is clear that asynchronous message creation causes a resource utilization problem which requires application level attention to avoid resource exhaustion. Universal user level management can become impossible in a complex system with multiple personalities and varied applications. Solutions could be constructed which required multilevel operations to reserve all the necessary storage before beginning but this sort of transaction processing has problems of its own and is, of course, inherently synchronous in nature. Client/Server synchronization can reduce kernel resource requirements to some small number of bytes per thread in the system. Management of application specific resource, of course, remains a potentially difficult problem but kernel level resource management for RPC might reasonably consist of nothing more than controlling the number of threads the system can have in existence at any one time.

4.1.3 RPC Specific Message Control Information Issues:

The message control structure associated with an RPC call refers to both the request and reply portion of the message. Besides the fact that it is the very essence of RPC to semantically link the send and receive, this arrangement turns out to be convenient with respect to the message buffer.

In the message library version of RPC, the message buffer format and in most cases buffer content does not change on reply. The message buffer represents the parameters sent by the original caller of the RPC. For many languages (including C) the parameters are not directly alterable by the caller. This means that in our implementation of RPC, it is not necessary to copy the message buffer back to the client. Absolutely requiring the format to be the same on reply makes it possible to always have one message control structure description of the message buffer instead of two. Having one message control structure describe both send and receive represents another useful compression of control information. Only one descriptor is necessary per parameter instead of two. The information associated with the receive and send side buffer disposition in the case of by-reference variables is kept separate, making decomposition of send and receive side specifics convenient.

There are two drawbacks to the coalescing of request and reply information. The first is the issue of verifying the compatibility of a client interface when the server is using override options to alter the disposition of local buffers associated with the call. In this case, the remote buffer disposition bits associated with one or more of the parameters of the incoming message are no longer valid. This problem has been gotten around by collecting all of the bits associated with server buffer disposition into a field. The server may check the incoming message control structure with by the same byte by byte comparison except for the addition of a masking operation before the comparison of parameter flags fields. The server is in full control of the compatibility check, based on the type and scope of the override, the mask may be used on all or some of the parameters of the incoming message. The second drawback is centered around the server_dealloc option. Special care will have to be taken when it comes to server_dealloc, the server may be compelled to check for this option and where it occurs send an override back on the reply. This is sub-optimal in the sense that if a client persists in sending messages with a server_dealloc on a parameter and the server persists in doing overrides in which the server_dealloc must be overridden. The server must continually send a reply message control structure and the message passing library 220 must on a call by call basis consult it. In the worst scenario, the server would check the incoming message control structure every time, doing a special check for the dealloc. This is not a large disadvantage over a hypothetical non-coalesced notion since in that case the server would have to send a message control structure with every reply. But it does require an extra check at user level on the part of the server and a cross comparison at the message passing library 220 level. This, of course, can be avoided by having the client send a message control structure that does not contain server dealloc, or through registration. The server can choose to register a message control structure which does not include the server_dealloc option and return the registration id for this to the client.

4.14 The Subclass of Supported Procedure Calls:

To be sure, when emulating procedure calls it is simply not possible to support the entire range of local procedure call behavior. Certain things are ruled out right away. There can be no side effects on global variables not associated with the parameters of the call but accessible to the caller and the callee via direct access. No inner scope tricks, either. A called procedure cannot act on a variable which has been declared as a local variable of the calling procedure or one of its ancestors unless that variable appears as parameter.

Beyond these obvious examples of side effects, however, lie a large set of perfectly valid calls which we do not support. The largest of these is multiple indirection of degree > 2. Though it could be supported, it was deemed not worth the trouble to support variables whose degree of indirection was above 2. Double indirection got the nod because it allows the callee to change pointer values, allowing the passing back of arrays.

In spite of the restrictions, the subset of procedure calls supported by RPC is large. It was a design goal to allow both uninterpreted message buffers built on the original caller's parameters and the transparent use of RPC. Clients do not have to send a destination port as the first parameter on their call. The message passing library 220 has the ability to send the destination port in the transmission data section. Subclassing could then be carried out through library swap or change in library resolution path. Further, clients are allowed to return data and pointers on their function calls. The message passing library 220 supports this through an optional separate status return for transmission status. Most importantly, buffer disposition classes have been set up to support a wide range of actions the called function might take. The client can fully expect the called procedure to remove a buffer after looking at the data, or allocate a buffer in which to return data. The range of supported semantics is determined by the buffer subclasses defined in Section 2. Beyond direct support, the client side proxy routine is capable of supporting local semantics, i.e., if the client was expecting the server to use a particular heap source when allocating a buffer, the proxy might allocate such a buffer using local calls and change the RPC call to reflect a write into this buffer. This would, of course, cause the proxy to rewrite the message buffer and would have some effect on performance.

Support of such a wide class of procedure calls was designed to ease the path towards co-residency. (Allowing callers and callees to be written to one paradigm and yet execute both remotely and locally without performance loss.) The local case was the most performance sensitive. In order to get the best possible performance here, the paradigm had to look as much as possible like a local procedure call. This was achieved, the calls can be indeed, and are in some cases (those on which the client does not send a destination port), local calls. The enumerated list of properties below along with the restrictions mentioned in the first two paragraphs of this section characterize the supported set of procedure calls. Procedures not venturing beyond these options are supported, even if they were never written to work in a message passing environment.

1. Function return values may be full word or pointers to structures.

2. By-reference data fields may be dynamic in size.

3. The callee may delete a buffer sent by the caller.

4. The callee may create a buffer and supply it to the client through the setting of a double indirect pointer.

5. The callee is capable of writing into a buffer supplied by a client and if that buffer is not big enough, either.

   a. returning an error

   b. truncating the data

   c. getting a new buffer and pointing a double indirect pointer to it.

6. The callee may push the data associated with different calling parameters into a single pooling buffer.

7. The callee may push data associated with different calling parameters into multiple pooling buffers.

There are some restrictions on ports when they are sent as arrays. The pointer to the array can of course not be more than double indirect and all the ports in the array must have the same disposition. Since ports are a data type special to message passing, these restrictions might be more properly looked upon as restrictions on the data type.

4.1.5 RPC Specific Use of the Transmission Control Information Component:

RPC uses the transmission control information subcomponent in the same was as other message passing models. It can alter defaults for NDR state, it can pass information back and forth between proxies via the trailer. The RPC, however, has some additional needs which must be met in order to support message passing transparency for its clients. The two major options are STATUS and DESTINATION PORT. The RPC subsystem supports the return of pointers and data on remote function calls. The default behavior for the message passing library 220 is to combine procedure return status and transmission status much as the CMU mach message passing service did. In order to separate the function return code information, the client side proxy must request the STATUS return option.

The function return status is then placed in a field in the optional header area. The transmission status is returned in the normal way. This preserves the message buffer, allowing it to appear just as the original caller's parameters did. The transmission section destination port override allows the proxy to determine the destination of the message without changing the message buffer. Again this is meant to support the dual notion of the fastest possible user level interface with support for transparent procedure call emulation. It seems less likely that 2 way IPC 122 for instance will want to hide the destination port notion from the caller, but the option remains available to all proxies. As with non-RPC uses, the trailer will prove useful for its security token, sequence number, scheduling information and possible routing information.

4.1.6 Priority Based Queuing, Making the Client Queue Pluggable:

The Capability engine 300 has generic interfaces for queuing and dequeuing of incoming messages. The routine actually called is determined by a field in the port structure of the target port. The Capability engine 300 consults this field and calls the procedure pointed to by it. The Capability engine 300 also has the interfaces that are called to alter this field. By setting up the proper queuing is the order of the day, the queuing code can check the schedulable entity associated with a blocked thread or a field in the message and queue or dequeue a thread/message accordingly. This is the basic idea behind the message passing library 220 and capability engine 300 support of multiple queuing methods. The capability sender queuing call is made (either through svc or directly) with a kernel object as parameter. The first portion of both the message and thread structure are kernel objects. The queuing procedure itself determines the type of the kernel object via a type field in the self defining data structure (the kernel object) and proceeds accordingly.

RPC, of course, does not queue messages and so message specific functionality in the queuing code would go unused. For details on the expected placement of scheduling information in message structure and the dual queuing of messages and blocked threads. The client queuing function of the capability engine 300 is not expected to be called directly in the case of RPC, rather, it is expected that SVC calls which encounter a shortage of server resource (either active thread or passive thread body notion) will trigger the capability engine 300 to call the queuing mechanism.

In Fig. 27, the Port is shown inside the capability engine 300 because ports are only accessible through capability calls.

4.1.7 Support For Message Server Spaces, Demultiplexing on Message ID:

It is often the case that a series of functions have enough features in common or contribute to a single purpose in such a way that there is advantage in describing them as a set. Further, if these functions share the same resource and information base it is important that the members of the set not be physically divided. In an effort to support this and to economize on ports, port level demultiplexing was carried forward from the CMU mach_msg model. (It is also necessary for compatibility.)

The message id appears as a field in the header, it determines the interface and format of a message amongst a series of interfaces associated with a single port. The message id is not a primitive of the message passing library 220 in that the library does not use its value in message handling decisions. It therefore could be relegated to the trailer as an option. However, it has been the overwhelming experience with CMU's mach_msg that demultiplexing RPC's are preferred over one port, one method or a more general and interpretive IPC 122 with its subsequent parsing costs. For a fixed set of interfaces, the message id is indeed an optimization of interpretive IPC 122 with its subsequent parsing costs. For a fixed set of interfaces, the message id is indeed an optimization of interpretive IPC 122. It is at once faster and more powerful. By convention, the message id transmits semantic as well as format information. (Interpretive IPC 122 here is defined to mean the message format is not known ahead of time and the message control structure must be consulted.)

In the demultiplexing model, the user level server is comprised of a primary procedure which places messages on and recovers messages from a port. This procedure does some general processing, (message too large, restart handling, buffer handling, etc.) and in turn, calls a server side proxy. The proxy called is determined by the message passing id. This server side proxy does the discrete function level specific setup and checking.

The general server loop does get involved in discrete message processing in one place. The message control structures are made available to it through a table indexed by message id. The byte by byte check code is generic. It is just the data involved which is function specific. Further, alterations to server side options are necessarily server wide in scope. It is the general server function that is the most appropriate place for the necessary adjustments to the server side format check. It is also true that the server side stubs tend to be automatically generated. This makes them a less convenient target for receive side buffer disposition customization. An outline of the flow of execution in a typical message receive/send, as shown in Fig. 28.

- Primary Server Function receives Message
- Primary Server Function checks status, i.e., message to large and does appropriate high level handling.
- If registered, primary server checks index table to relate registration id to message id.
- If not registered and client not trusted, Primary Server Function uses message id to get message control structure template and check against incoming (obviously requested) sender message control structure.
- Primary Server Function uses message id as offset into table to get the proper proxy function. Primary Server calls proxy function.
- Proxy function does any necessary transformations on incoming data. These transformations are function/application specific and outside of the message passing architecture except for their support in automatic proxy generation tools.
- Proxy function calls the targeted endpoint. (the callee)
- Proxy function does function specific cleanup, including any data transformations. Proxy function returns.
- Primary Server Function re-works header fields, it is not allowed to increase the size of the header unless another buffer is used. (There may be server temporary data 400 to be sent on the reply below the header in the receive buffer.) The primary server optionally includes a reply message control structure (rare) and reworks the message returned by the proxy to reduce its size (rare). (Such customizations may or may not be supported directly in the product, the application writer may be left to customize the server loop and data structures by hand.)
- The Primary Server Function calls the message passing library 220 with a send/rcv. The supplied header is pointing at the reply structures. (The message buffer which in turn is pointing at temporary and permanent fields that contain data to be supplied on the reply.) The header is at the top of the receive buffer. The receive buffer is big enough to hold any of the incoming message headers and their temporary data 400, or one of the oversize options may be encountered.

4.1.7.1 Dynamic Message Server Spaces:

The support of dynamic linking and co-residency is very powerful. It allows the download and linking of routines into a target space. Proper implementation will allow a downloaded function to connect to a local procedure call and perform possibly as a local procedure call without any additional overhead, effectively bypassing client proxy, server, and server proxy routines. When the function call is aware of message passing, it will still be necessary in the local case to have a proxy inserted between the caller and callee, but the complexity and overhead of this proxy will be greatly reduced when contrasted with a remote call.

Co-residency also supports the remote setup of servers. To support this, co-residency must go beyond simple download and link functionality. In the case of a pre-existing server, download and link from an external source could be used to alter one or more of the server proxies and their endpoint routines. In order to do this, however, the remote entity would need to know the name of the proxy, possibly the name of the endpoint and have general write permission, i.e, the task port for the target task. Just to support this functionality with some degree of protection, a complicated set of user level utilities with would have to be created. These utilities would be trusted and a target task would entrust its task port to them. Other tasks wishing to download function would have to communicate with the download target through these utilities.

Even if the complicated application level tools were acceptable, the level of functionality really isn't sufficient. Additional function requires a high level of communication between the target and the task attempting remote download. The caller cannot start a server, or add a new message id to an existing server without some method outside of the defined notion of co-residency.

In order to support these notions in a simple straight forward manner, we need support for a dynamic server model. A task wishing to make itself available as a dynamic server must create and export a port which makes the series of server creation, manipulation, and shutdown routines available. A server for servers. This server/server exports the calls presented by the server library. The default server loop is not just a shared library routine. The server_create call creates a threadless instance of a server and returns a handle. This handle is used by subsequent calls to change optional aspects of the server instance, add or delete server threads, associate proxies and by consequence their endpoints, add or remove receive buffers, or shutdown and clean up the server instance. After using basic co_resident utilities to download specified code into a target task, the remote caller would send a server_create message to the server/server port and receive a handle back on the reply. The caller may have supplied a set of proxies on the call or may fill in the proxies through subsequent calls. The caller has an additional call which is not one of the calls exported by the server package. An extra call is needed to create a thread and then direct that thread to associate itself with the target server instance. In the passive model, it is possible to simply provide the thread body resources to the receiver, but in the active model, the server acquires threads via a call by the target thread. There is an advantage to having the routine built in this way. The target server task is free to adjust post processing or customize thread state or resource for its specific needs. Because of the notion of server instance, a server persists even if its threads exit the server. In this way, exceptional conditions can cause a thread to return from its run_server call. The task is then able to customize exceptional processing. The thread can then be returned to the server loop. If the exception the thread is returned on is a simple return_server_thread, the thread is free to re-associate itself with the server, run some other unrelated task or self-terminate.

4.1.8 Anonymous Reply Support:

In the message passing library 220, the semantic link established between request and reply regarding data, buffer disposition and message buffer format is separated from the execution path and the resource necessary to carry out the request and reply. Through a separate set of fields in the message control structure, options on the port, and thread support options in the tasks themselves, the resource used to carry out the request and reply is explicitly manipulated. In the simplest, fastest case, there is no need for an explicit reply port. The client is simply blocked waiting for the completion of the remote procedure call, the server or at least a thread of the server is dedicated for the duration of the call to completing the remote procedure call and returning the result. The simple case, provides the message passing library 220 with an opportunity to employ the same technique used to optimize by-reference data transfer in effect, by-passing the explicit port wait that occurs in asynchronous and explicit receives. The message passing library 220 can in this case, avoid the expense of contacting the capability engine 300 both to do the wait and to map a send or send_once right into the server's space. There are cases, however, where for throughput or transmission control reasons more flexibility is needed.

Because of this flexibility, in some circumstances, an explicit reply port is required on either the server or client side in order to keep track of the reply target. Though rare, the client may wish to declare an explicit reply port in order to allow for intermediate message delivery. The proxy routine would in this case be capable of receiving these intermediate messages, processing them and then re-establishing the wait for reply by doing a receive on the explicit reply port. An example of this behavior can be found in the receive side handling of thread_abort_notify.

Some application environments accept asynchronous signalling only at the time of service calls. Even though the system is not strictly asynchronous, the asynchronous signals may need to be received within a bounded time. This can be partially determined by the frequency at which service calls are made by the target code, the target doing null service calls when the execution of local code does not include real service calls at short enough intervals. The delays during these service calls, however, might be longer than the system can tolerate. In this case, it must be possible to abort the target out of a block on send (or request), a block on receive (or reply) and possibly out of server processing via an out of band abort signal to the application server routine. If the client side proxy is set up to handle it, the send side abort with signal is straightforward. The client awakes with an abort_notify signal, processes it and if it wants, restarts the RPC. If the server is already processing the request, however, the client is waiting on a reply, in order to receive a thread_abort_notify signal during this period, the client has to have undertaken the RPC with an explicit reply port. In this way, the message passing library 220 can send an abort_notify message to the client and the client can re-establish its wait on the reply. If the client did not supply an explicit reply port, the message passing system will pend the abort_notify state and include it with the reply coming back from the server.

In order to avoid an explicit reply port on the server side, the server must be able to guarantee that the thread sending back the reply will be the same one that was associated with the request. In this way, the client awaiting a reply can be registered in a structure associated with the server thread structure. The server may not be able to guarantee this as it may be subject to a user level threading package and as a result subject to some form of thread multiplexing at user level. Such multiplexing is often done in an effort to support throughput, real-time, or some form of explicit serialization.

Seamless support of anonymous reply port optimization requires that a client decision with respect to reply port be hidden from the server and visa versa. The message passing library 220 achieves this with the algorithm shown in Fig. 29. The algorithm starts at the point that the message passing system has both the client send and the server receive in juxtaposition. This is, of course, always achieved in RPC.

Case 1, of course, gives the best performance. However, case 3 should perform better than either 2 or 4 because it is not necessary to create and place a port right in the server's space. Case 3 may perform nominally better than case 4 because the anonymous port is a light weight affair, not requiring the state and setup of the normal port types.

Upon return from the request, the server thread's data structure is checked for an outstanding reply. This will be present in case 1 and case 3 above. If this is an example of case 3, a second field in the port structure points to the client reply port. If the client is blocked on a port, it is removed. If it is not blocked, the server is made to wait on the port as a sender. When the client is available, the reply is delivered, the client thread returns and the server resources or thread are free to take on another message.

If the server's thread structure does not point at the client, there must be an explicit port in the remote port field of the server message call or an error is returned to the server. The client thread is retrieved from this port if it is there and the transfer proceeds. If it is not on the port, the server thread block awaiting it.

4.1.9 ABORT Support:

Abort support is a complex issue made more complex by the fact that abort support is not one function but three. Standard thread_abort will abort an ongoing message or wait without regard to restartability. Its use is limited, therefore, to drastic situations such as thread termination or at least termination of the stream of execution utilizing the thread. The second form of abort is thread_abort_safely, which should probably be called thread_abort_checkpoint or have some other better suited moniker. Thread_abort_safely's real purpose in life is to bring the target thread quickly to a state where a signal can be safely given. The signal is asynchronous and its mechanism must not be detectable to the synchronous stream of execution. It is, therefore, imperative that Thread_abort_safely be restartable. The third form of abort, Thread_abort_notify like thread_abort_safely is used for signalling. Unlike Thread_abort_safely, Thread_abort_notify delivers its signal directly to the execution stream of the target thread via return status on a message passing call. Its purpose is not to guarantee a quick restartable return from a kernel call so that a user level signal processing routine can be called and returned from. If the thread is running at user level, it posts a notify state and bides its time. Thread_abort_notify can only delivery a signal on return, restartably aborted or otherwise, from a message passing call.

The aims of the three types of abort are different enough that they influence their implementation, therefore they will be considered separately.

4.1.9.1 Thread_Abort:

There are two types of wait that a thread is capable of exercising, a kernel based one and an external server based one. In the case where the caller of abort is not worried about thread restartability, the only important considerations in waking up a waiting thread are the server or kernel resources and state. The thread may be returned to user level with a thread_aborted declaration at any time so long as the server/kernel are not left in an undefined state or with an orphaned resource. In the kernel it should be possible to either clean up the resources synchronously or less-desirably, create a care taker. In fact, in the new, more modular microkernel 120 architecture, there may be no kernel waits other than waits on message passing send and receive. In any event, the exact disposition of kernel resource recovery is beyond the scope of a paper on message passing. The server case, however, directly involves the message passing system.

In RPC, the abort function may find a thread either blocked waiting to begin a request or blocked awaiting a reply. If the thread is blocked on the request, abort is simple and restartable, return the thread with request_aborted status. The server was not yet aware of the request and no recovery action at all is required. If the thread is awaiting a reply the situation is much more complicated.

In the case of Thread_abort, an attempt may be made to stop the server as soon as possible rather than letting it complete a now useless bit of work. The first attempt to abort the server is made via the port. A field of the port structure points to an abort_notify function. If the server wishes to support early termination of work for an aborted client, it may choose this method. The message passing library 220 passes a message to the abort notification port containing the port, and the sequence number of the associated message. (The port is necessary because the message may have been delivered on a port set.) In any case, the state of the port awaiting the reply will be altered such that when the reply is sent back, the message will be destroyed and the server reply apparatus liberated. If the port is destroyed first, the server will simply encounter a dead name for the reply port and may act to destroy the reply and continue.

If the message passing library 220 finds that the abort notify field of the receive port has not been filled in, it checks to see if the server requested anonymous reply port. If it did, the server has guaranteed that there is an unbreakable link between a specific server thread and the request. In the server anonymous reply case, the message passing library 220 executes a thread_abort_safely on the server thread and sends a signal indicating that the message being processed is no longer important. The anonymous reply port, if present, is destroyed. If the client sent an explicit reply port then the state of the reply port is set such that the reply message will be destroyed and the server reply apparatus liberated as if the reply was sent.

The client will return from its message passing call with a Thread_abort status. This status indicates to the client that the message was aborted and the associated resources and data lost. For systems which wish to use the less refined Thread_abort_safely, retry can be achieved if the client checkpoints its data before attempting message passing. The server state is important only if the server maintains state between invocations. In this case, the designer must insure that the server receives notification of client aborts and takes appropriate action.

From a real-time perspective, there is a danger to proper scheduling of resources in the case where the server acquired the scheduling properties of the client. From a scheduling standpoint, this is effectively the server passive model where the client entity runs in server space. After experience of an abort. The client thread is effectively cloned with one running temporarily in the server and one running in the client. If the priority of the client is high enough, the server thread (in the abort/signal scenario) might run to completion before encountering the signal to terminate. In the server explicit reply case when there is no abort notify port, there is no attempt to notify the server of a client abort.

It is only in the case of the abort notify port that the server, through the system designer, can ensure timely delivery of the client abort notification. If the active thread on the server abort notify port is given high priority or if the passive scheduling parameter assigned by the message passing library 220 is of high priority, it will be scheduled before and may preempt the client message processing. The server may then set user level state to communicate with the client message processing thread that it must terminate early.

4.1.9.2 Thread_Abort_Safely:

Unlike Thread_abort, the purpose of Thread_abort_safely is not to logically abort an ongoing execution stream, it is merely to interrupt it. Thread_abort_safely needs to get the target thread to a state where a user level routine can be run in order to deliver an asynchronous signal. It then must recover in a way which is transparent to the synchronous execution stream. In CMU mach_msg, thread abort resulted in a return to the mach)msg call with thread)abort)send or thread_abort_rcv. These calls were restartable, but a small interpretive loop and an extra procedure call were required to check for and carry out the restart.

In the message library, there is no user level retry worked into the message passing call. Thread_abort_safely and Thread_signal work together in such a way that the return stack is set to deliver the exception message and when the exception routine returns, a trap back into the kernel occurs. The return from exception trap then checks the thread structure and determines what queue it was waiting on and sets it back in its place. It is not presently planned to have a compatibility option to bring back thread_abort_send and thread_abort_rcv to user level. If absolutely required, it can be brought back, but when active, it will result in scheduling artifacts and inefficiency avoided in the method outlined below. Request and Reply would have to be ripped from their queues, causing the client to lose its place in the request queue and requiring expensive calls to the capability engine 300 to talk to port queues.

Thread_abort_safely is undetectable when it aborts a message from the request wait just as in the Thread_abort case above. It is also undetectable in the abort from reply wait case. In fact, the client thread is not effectively removed from either the request or the reply waits. The resources necessary to carry out the request remain on the queue when a client experiences a Thread_abort_safely. In the case of the active server, the server thread is free to pick up this request even during an ongoing Thread_abort_safely. In the case of the passive server model, unless otherwise instructed (see real time considerations below) a shuttle is cloned and the server processes the request. Another way to look at it is that when Thread_abort_safely is experienced. The RPC is separated from the thread body of the client. This thread_body is then endowed with a shuttle and told to execute an exception message routine. When the exception message routine returns, it returns to the kernel. The message passing library 220 then removes the shuttle and re-establishes the client thread connection to the RPC. In the meantime, the entire RPC might have occurred, including the placement of reply resource in the client space. There is a precise corollary for the active model except that it does not suffer from the scheduling difficulties.

A link is kept with the RPC even during the Thread_abort_safely call, should the thread be scheduled for termination, probably through a Thread_abort, the RPC is reachable and the reply port and send request will behave as described above in section 4.1.9.1. The server resources are guaranteed not to become permanently pinned on a zombied reply port.

There are several ways to view the real time issue with respect to Thread_abort_safely. It could be argued that the asynchronous signalling process is a schedulable event in its own right. The exception message port would carry with it a scheduling priority apart from the scheduling information associated with the thread targeted for the signal. In this model, no action need be taken regarding the ongoing RPC. If the signal is viewed as an act carried out by the target thread itself, however, there is a need to adjust the scheduling information of the RPC at least in the passive server case. If the RPC has not had its request considered, the scheduling info can be altered to reflect suspend. This, of course, may effect the order of queued requests processed. If the request is already underway and the client abort notify port is active, the message service can send a message to the server stating that the request should be suspended. If the client notify port is not active and the server is using anonymous reply, the server thread may be suspended. It is assumed that the first non-intervention approach will win the broadest appeal as any other approach will affect completion time, and this is in conflict with the attempt to make Thread_abort_safely transparent to the synchronous execution path.

4.1.9.3 Thread_Abort_Notify:

As mentioned earlier, the main objective of Thread_abort_notify is to deliver an informationless signal to the caller of the message passing service upon return from a message passing call. In an effort to deliver the signal in a timely fashion, the call may be aborted, but only in a restartable fashion. Thread_abort_notify only delivers signals as part of the return status from message passing requests. For this reason, if a Thread_abort_notify is sent to a thread, either not waiting on a message queue or at user level, the thread is put in notify_signal state and action is delayed until it reaches a state where notification can be delivered.

Because this notification method does indeed involve aborts, it is not possible to avoid completion time anomalies on the RPC as in the thread_abort_safely case above. This is a direct consequence of the need to expose the signal to the synchronous execution stream. Because Abort_notify is visible to the synchronous execution stream and because it is not a mainstream function. An option to ignore thread_abort_notify has been included in the message header.

Thread_abort_notify when experienced by a thread blocked on the request queue, results in the request being removed from the queue and the client receiving a thread_abort_notify_send message. There are no artifacts, the server was never aware of the request, and the client is free to retry the RPC or not based on its handling of the notify signal directed at it.

When Thread_abort_notify aborts a wait on reply, the client thread is taken off of the reply wait queue and returned with a status of thread_abort_notify_receive. The client is then free to process the notification and do a receive on the reply port to continue the RPC. Thread_abort_notify will not abort a wait on reply unless the client has made its request using an explicit reply port. This was done because returning the client while it was waiting on an anonymous reply would either have required the client to return through a special trap instead of doing a receive on the reply port or a receive right to the anonymous reply port would have to have been placed in the client's space.

If the client does indeed supply an explicit reply, the system designer may feel compelled to take some action to effect the scheduling of the server request handling of the associated RPC, especially in the case of the passive server model. The case differs from Thread_abort_safely in that the synchronous code path is activated. The code is free to go off and do anything, delaying its return to wait for the reply indefinitely. It is not possible to protect the system from this behavior as it is with the separate signal handling code path of Thread_abort_safely. For this reason, abort on reply will also be suppressed by the message passing library 220 unless either the sender is using the anonymous reply port option, in which case the server thread may be signalled, or the client abort notify port is active so that a client abort message (of notify flavor) can be sent to the server.

4.1.10 Shared memory support 800:

Shared memory regions may be established through the message passing library 220 via two means. 1: The explicit shared memory by-reference parameter with matching overwrite on the server side to establish the shared region; or 2: The passing of a share capability. Though either method will establish a region, the first is considered more appropriate for RPC and will be the only case described in detail in this section. Passage of a share capability is less constraining to a server. The server is free to send the capability on to another task and if the write was a send right instead of a send once, the server may share the share region with others. This will create a common buffer among multiple parties. The server does this by doing a copy send of the capability in a message passed to a third party before consuming the original right in a mapping call. Though the forwarding of data on message receipt will occur with the sending of a message, broadcast is not defined for RPC and only the target server will receive an overt signal that the message buffer has been filled or freed. Capabilities cannot be established on shared regions, we are, therefore, protected from broadcast in the explicit shared memory setup case.

The vast majority of servers are stateless, that is they do not hold information acquired specifically from the data associated with a request beyond the delivery of the associated reply. (This, of course, does not include stochastic information concerning frequency of usage and resource requirements.) Because of the preponderance of stateless servers, it is expected that cases of shared memory usage in RPC will strongly favor client passing of control on the request and recovery of control on the reply. The client may either pass information in on the request or recover it on the reply or both. Fig. 30 are diagrams outlining the steps undertaken to establish shared memory and the protocol for passing data back and forth.

The establishment of a shared memory region involves the message passing library 220 creating data structures to hold information regarding the size and location of the share region in the local space and details on the whereabouts of the region in the remote task. The information is attached to the task structure of both parties. One final warning: Multiple mappings, mapping the region between more than two tasks, results in a linked list of share data structures which must be parsed on each share region pass. These should be avoided unless there is an express need for broadcast.

It is important to note that a shared memory region cannot be created without the active support and knowledge of both the sender and the receiver. Both the sender and the receiver will be aware of the size and location of the shared region within their own space. In this way, if a server did not find that a particular client was trusted enough, it could decline to accept a client directed share. The shared regions are also directional in the sense that one party establishes them and then offers them up to be shared. This is important because it is often true in the active server case that a server cannot trust the backing pager of a region of shared memory offered by a client. The associated pager might be inactive and cause the server to hang on a memory fault. In this case, it must be the server who offers a region. The region, of course, will be backed by a pager that the server trusts. Because of real time considerations, client pager issues notwithstanding, it is likely that the client directed share will be the method of choice in the passive server model. The passive model asserts that the client thread of execution enters the server space. In this case, it is the server pager that the client might not trust. Should the thread associated with the client request hang due to pager inadequacy, the thread body resources associated with the call could be reclaimed through the same mechanism used to prevent priority inversion on blocking requests for other resources.

Message control structure allows the client to send the share buffer either statically, where its size is described in the message control structure or dynamically, where another parameter on the call serves to give the size. Overwrite options give the receiver the ability to direct the overwrite region to a specified region of mapped memory or allow it to be placed in formerly unmapped memory.

In Fig. 31, Message Passing library 220 does not see overwrite buffer on server side; it checks to see if client region is actually shared by server by checking task specific state info setup on share region initialization. If region shared, it checks to see if physical share; if so, just pass on size and translated address. If not, do the explicit copy of data. If not shared, return an error to client.

The message passing library 220 is very flexible with respect to the usage of a shared memory region. Either party may send data in it. Either party may initiate the call, i.e., if a client existed in the server task space which called a server in the client's space, the message passing library 220 would support it. Further, all or part of a shared region may be involved in a transfer. The client may start a transfer on any address within a shared region and end on any address up to the last one contained in the shared region. Messages may contain multiple areas from the same shared region in separate parameters. Data from multiple shared regions may be passed in the same message.

4.1.11 Support of One-Way Messages:

One way message support in an RPC system is apt to strike one as an oxymoron. There are, however, conditions which require support of ne-way messages, especially on the server side. In the active server paradigm, the server thread usually performs a reply with a subsequent receive for the next message. This is fine for the steady state condition, but how do you start it? The answer is that you start the server with a one way receive. The thread resource is put to sleep awaiting the arrival of the first message.

Server's may find it convenient to process subsequent messages while a particular message is blocked, waiting for a resource. Further, they may wish to use the thread resource associated with the blocked message. They are allowed to do this in the explicit reply port case. They will have to do a receive without a reply to get the next message. When the blocked message is finally reactivated, the server finds that it must now do a one-way send to institute the reply.

There are still RPC semantics associated with these one-way messages. Message control structure information, if sent with the reply, must be in the RPC coalesced form. Both in the reply and the request case, the transfer is fundamentally under the control of the client side message control structure as outlined in section 4.1.3.

One-way RPC send is not supported on the client side, and were it not for thread_abort_notify, it would not be necessary to support client one_way RPC receive. When the client is aborted out of a wait on reply in the thread_abort_notify case, a thread_abort_notify_receive signal is returned to the client. The client then must explicitly re-establish the wait for reply. The requirement to explicitly re-establish a wait for reply will also be experienced if it becomes necessary to support the old thread_abort_safely option to return thread_abort_send and thread_abort_rcv as direct signals to the client. (This might be required for compatibility with old interfaces.) It would then be necessary for the client code to explicitly re-establish the reply wait as in the thread_abort_notify case.

4.1.12 The Role of an Interface Generator:

The RPC implementation of the message passing library 220 was meant to support RPC without the need of compiled front ends. The interface presented by the message passing library 220 allows proxy routines to support RPC without expensive run-time interpretation. This is not to say that the message passing library 220 will not support front ends, compiled or otherwise, but RPC can be supported with simple message control structure creation routines.

It is fully expected that the RPC interface will run with a variety of front ends. The MIG tool has been ported to the message passing library 220 for backward compatibility. It has been extended to support the increased functionality. Porting of the MIG front end has proven that the new RPC is a superset of existing functionality as exposed by the MIG interface. Other front ends may be ported to provide compatibility with existing application code bases.

Because the message passing library 220 supports a full RPC interface, a front end need not be a context free grammar. It may be just a finite state automata. This greatly simplifies front end compiler design and code. Where another model or an extended model is desired, however, there are several places where performance on some machines might be improved through front end manipulation of the message buffer contents.

The message passing library 220 will accept double indirection pointers but under some circumstances (i.e., some machines might be extremely slow at de-referencing pointers across a protection domain boundary), it might be better to de-reference the pointer in user space possibly even twice, making it direct data. This would require the message option prescribing the writing back of direct data to the client. Dynamic data fields could be turned into static data fields. This would require the message control structure be written with each pass since it is here that static buffer lengths are kept, but it would remove the need for a separate count parameter. Finally, all direct parameters could be coalesced into a single direct data field, reducing the parsing time necessary in the message passing library 220.

It is not expected that any of these options will actually produce better performance in any but the rarest cases, but they are mentioned in order to show the flexibility of the RPC interface.

The RPC interface does not collect server interfaces together or create the proxy function tables and message control structure tables. It is convenient to have a front end that can automate this. A front end may also prove the proper place to do data transformations. And most importantly, front end support should be coordinated with co-residency requirements, making the production of remote and local proxy libraries as automated as possible.

Although a specific embodiment of the invention has been disclosed, it will be understood by those having skill in the art that changes can be made to that specific embodiment without departing from the spirit and scope of the invention.

Claims

1. A method for interprocess communication in a microkernel architecture data processing system, comprising the steps of:

   loading a microkernel including a capability engine module into a memory of the data processing system;

   forming with said microkernel a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object;

   forming with said microkernel a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights;

   registering in said capability engine, said first set of port rights for said first task container and said second set of port rights for said second task container;

   comparing said first set of port rights and said second set of port rights in said capability engine;

   and enabling a transfer with said capability engine, of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object.

2. A method as claimed in claim 1, wherein: said data processing system comprising in a shared memory multiprocessor system;

   including a first and a second processor;

   said first task container has a thread that runs on said first processor;

   said second task container has a thread that runs on said second processor and the enabling step enables said first processor to communicate with said second processor.

3. A method as claimed in claim 2, wherein: the first and second processors, each run a microkernel operating system; the method comprising the steps of:

loading a first microkernel into a memory of a first processor in a distributed data processing system, said first microkernel including capability engine module;

forming with said first microkernel a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object, said first task container having a thread that runs on said first processor;

forming with said first microkernel a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights, said second task container having a thread that runs on an I/O communications processor, said I/O processor coupled by a communications link to a second data processor running a second microkernel operating system;

enabling a transfer with said capability engine, of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object, thereby enabling said first processor to communicate with said I/O communications processor;

transferring said pointer and said first port rights from said I/O communications processor to said second processor to confer onto said second processor said capability to access said memory object.

4. A subsystem for interprocess communication in a microkernel architecture data processing system, comprising:

a microkernel in a memory of a data processing system, including a capability engine module;

a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object;

a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights;

said capability engine registering said first set of port rights for said first task container and said second set of port rights for said second task container;

said capability engine comparing said first set of port rights and said second set of port rights; and

said capability enabling a transfer of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object.

5. A subsystem as claimed in claim 4 wherein the data processing system comprises in a shared memory multiprocessor system running a microkernel operating system, the multiprocessor system having

a first processor and a second processor sharing a memory said second task container has a thread that runs on said second processor;

said capability engine enables said first processor to communicate with said second processor.

6. A subsystem as claimed in claim 5 wherein:

the first and second processors are coupled by a communications link and each run a microkernel operating system;

a first microkernel in a memory of said first processor, said first microkernel including a capability engine module;

a first task container in said memory having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task container to access said memory object, said first task container having a thread that runs on said first processor;

an I/O processor sharing said memory with said first processor, and coupled by said communications link with said second processor;

a second task container in said memory having a set of attributes defining a second communication port and a second set of port rights, said second task container having a thread that runs on said I/O communications processor; capability engine registering said first set of port rights for said first task container and said second set of port rights for said second task container; said capability engine comparing said first set of port rights and said second set of port rights; said capability engine enabling a transfer of said pointer and said first port rights from said first task container to said second task container to confer onto said second task container said capability to access said memory object, thereby enabling said first processor to communicate with said I/O communications processor; said second processor running a second microkernel operating system; and said I/O communications processor transferring said pointer and said first port rights to said second processor to confer onto said second processor said capability to access said memory object.

7. A system for interprocess communication in a microkernel architecture, comprising:
a memory means in a data processing system, for storing data and programmed instructions;
an interprocess communications means in said memory means, for coordinating message passing between tasks in said memory means;
a first task in said memory means having a set of attributes defining a first communication port and a first set of port rights, and having a pointer to a memory object, said first set of port rights conferring a capability on said first task to access said memory object;
a second task in said memory means having a set of attributes defining a second communication port and a second set of port rights;
a processor means coupled to said memory means, for executing said programmed instructions;
a first thread in said memory means associated with said first task, for providing said programmed instructions for execution in said processor means;
a capability engine means in said interprocess communications means, for registering said first set of port rights for said first task and said second set of port rights for said second task;
said thread providing a message from said first task to said interprocess communications means, for providing said pointer to said second task;
said capability engine means comparing said first set of port rights and said second set of port rights; and said capability engine means enabling a transfer of said pointer and said first port rights from said first task to said second task to confer onto said second task said capability to access said memory object.

8. The system of claim 7, which further comprises:
said first task represents an application program.

9. The system of claim 7, which further comprises:
said first task represents an operating system personality program.

10. The system of claim 7, which further comprises:
said first task represents a personality-neutral services program.

11. The system for interprocess communication in a microkernel architecture of claim 19, which further comprises:
a second processor means coupled to said memory means, for executing said programmed instructions;
a second thread in said memory means associated with said second task, for providing said programmed instructions for execution in said second processor means.

12. The system of claim 7, which further comprises:
said memory means and said processor means being in a first host system of a distributed processor system;
a communications link, for coupling said processor means in said first host system to a second host system of said distributed processor system;
a second processor means in said second host system, coupled to said processor means in said first host system over said communications link, for exchanging a reference to said pointer over said communications link.

FIG. 1

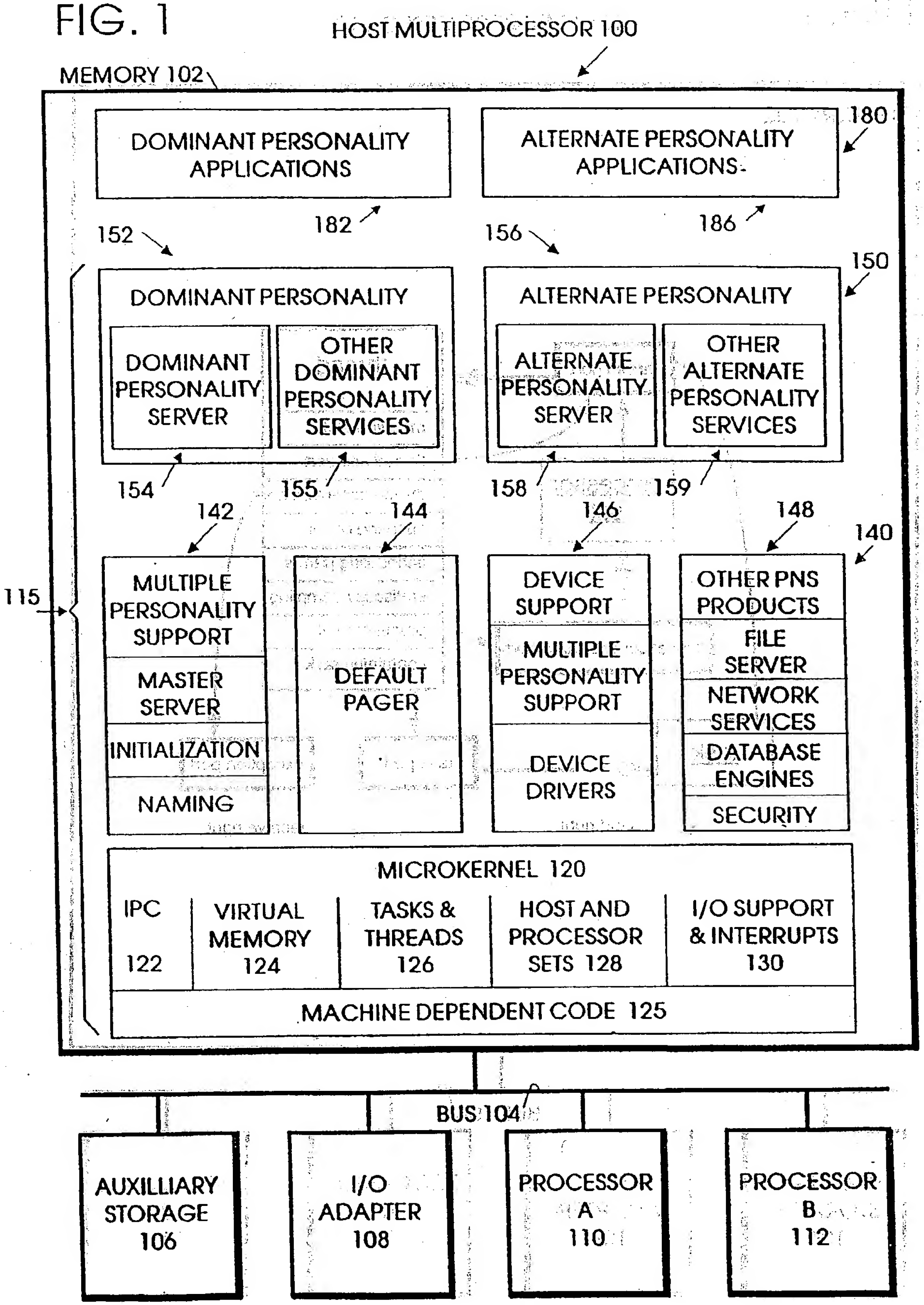

HOST MULTIPROCESSOR 100
MEMORY 102
DOMINANT PERSONALITY APPLICATIONS
ALTERNATE PERSONALITY APPLICATIONS
180
152
182
156
186
150
DOMINANT PERSONALITY
DOMINANT PERSONALITY SERVER
OTHER DOMINANT PERSONALITY SERVICES
ALTERNATE PERSONALITY
ALTERNATE PERSONALITY SERVER
OTHER ALTERNATE PERSONALITY SERVICES
154
155
158
159
142
144
146
148
140
115
MULTIPLE PERSONALITY SUPPORT
MASTER SERVER
INITIALIZATION
NAMING
DEFAULT PAGER
DEVICE SUPPORT
MULTIPLE PERSONALITY SUPPORT
DEVICE DRIVERS
OTHER PNS PRODUCTS
FILE SERVER
NETWORK SERVICES
DATABASE ENGINES
SECURITY
MICROKERNEL 120
IPC 122
VIRTUAL MEMORY 124
TASKS & THREADS 126
HOST AND PROCESSOR SETS 128
I/O SUPPORT & INTERRUPTS 130
MACHINE DEPENDENT CODE 125
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 2

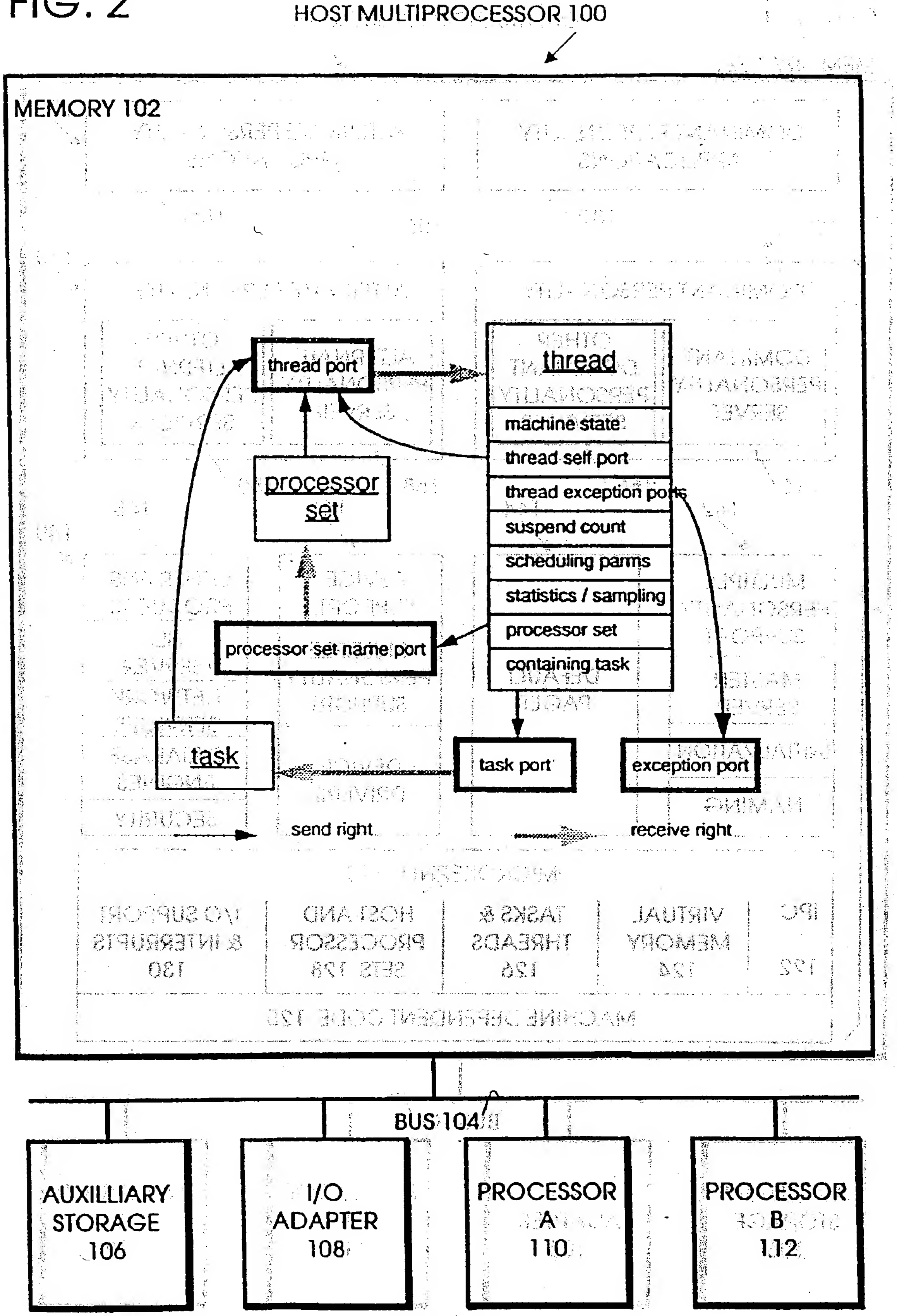

HOST MULTIPROCESSOR 100
MEMORY 102
thread port
thread
machine state
thread self port
thread exception ports
suspend count
scheduling parms
statistics / sampling
processor set
containing task
processor set
processor set name port
task
task port
exception port
send right
receive right
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 3

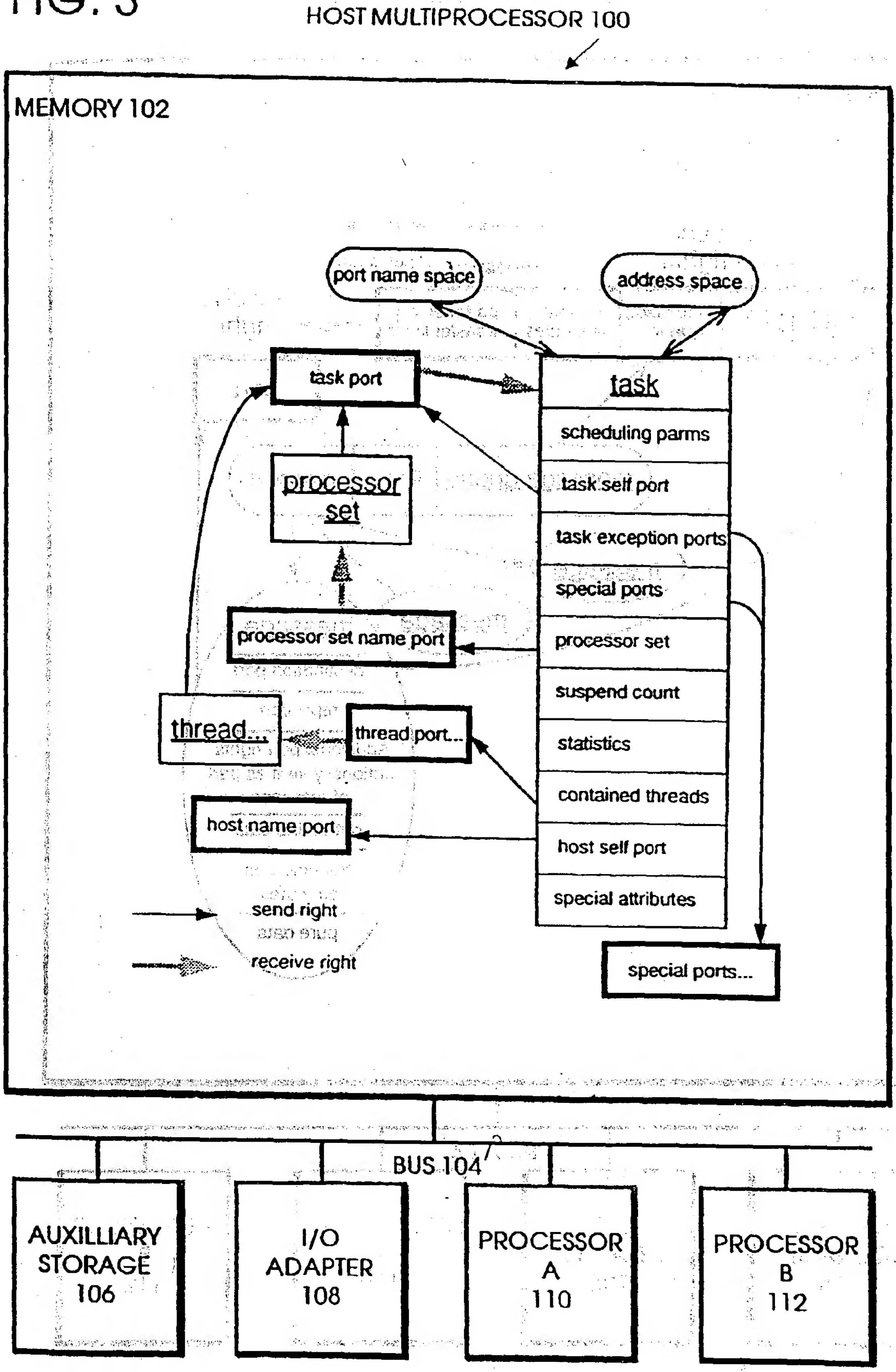

HOST MULTIPROCESSOR 100
MEMORY 102
port name space
address space
task port
task
scheduling parms
task self port
task exception ports
special ports
processor set
suspend count
statistics
contained threads
host self port
special attributes
processor set
processor set name port
thread...
thread port...
host name port
special ports...
send right
receive right
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 4

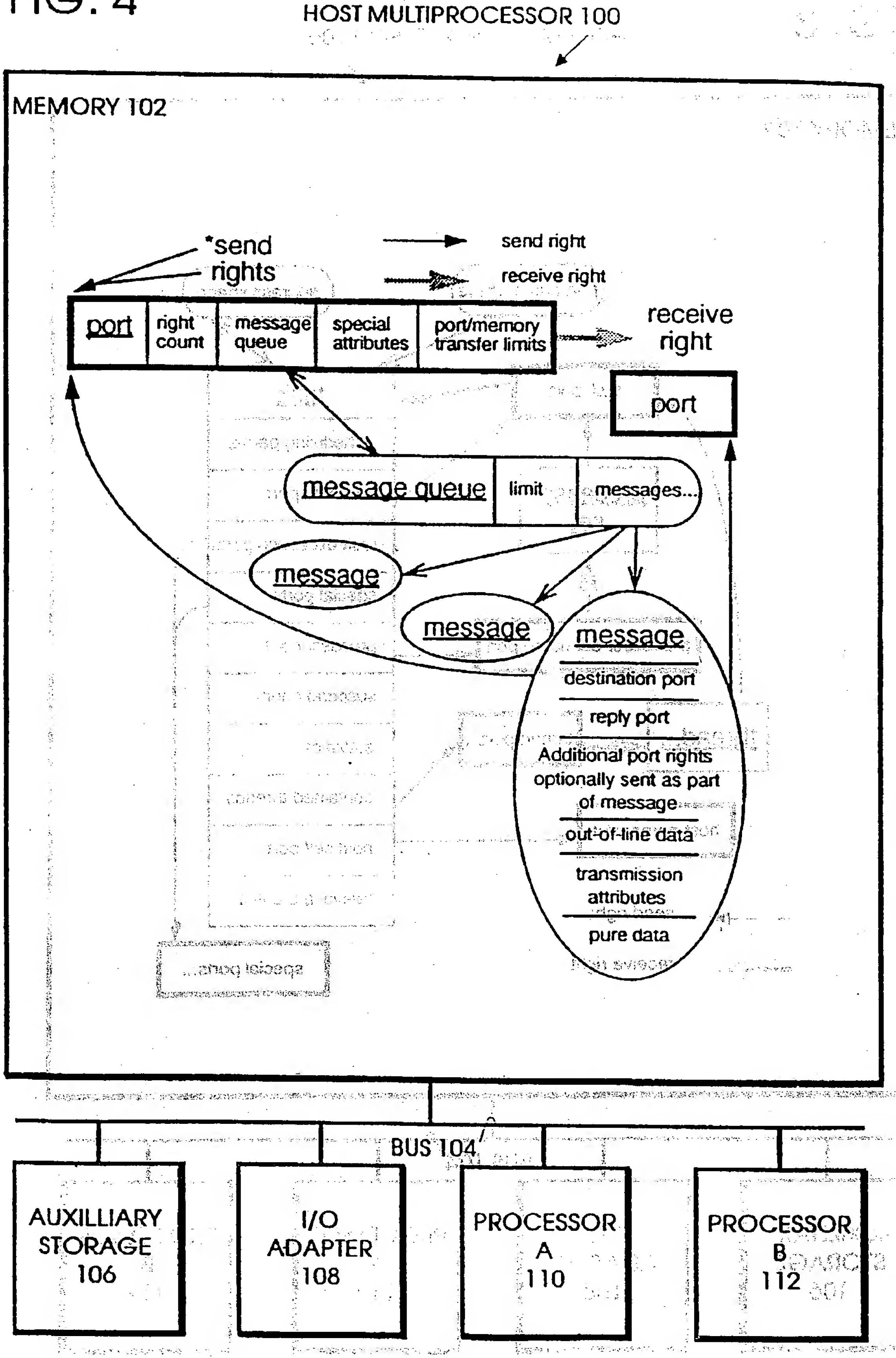

HOST MULTIPROCESSOR 100
MEMORY 102
*send rights
send right
receive right
port
right count
message queue
special attributes
port/memory transfer limits
receive right
port
message queue
limit
messages...
message
message
message
destination port
reply port
Additional port rights optionally sent as part of message
out-of-line data
transmission attributes
pure data
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 5

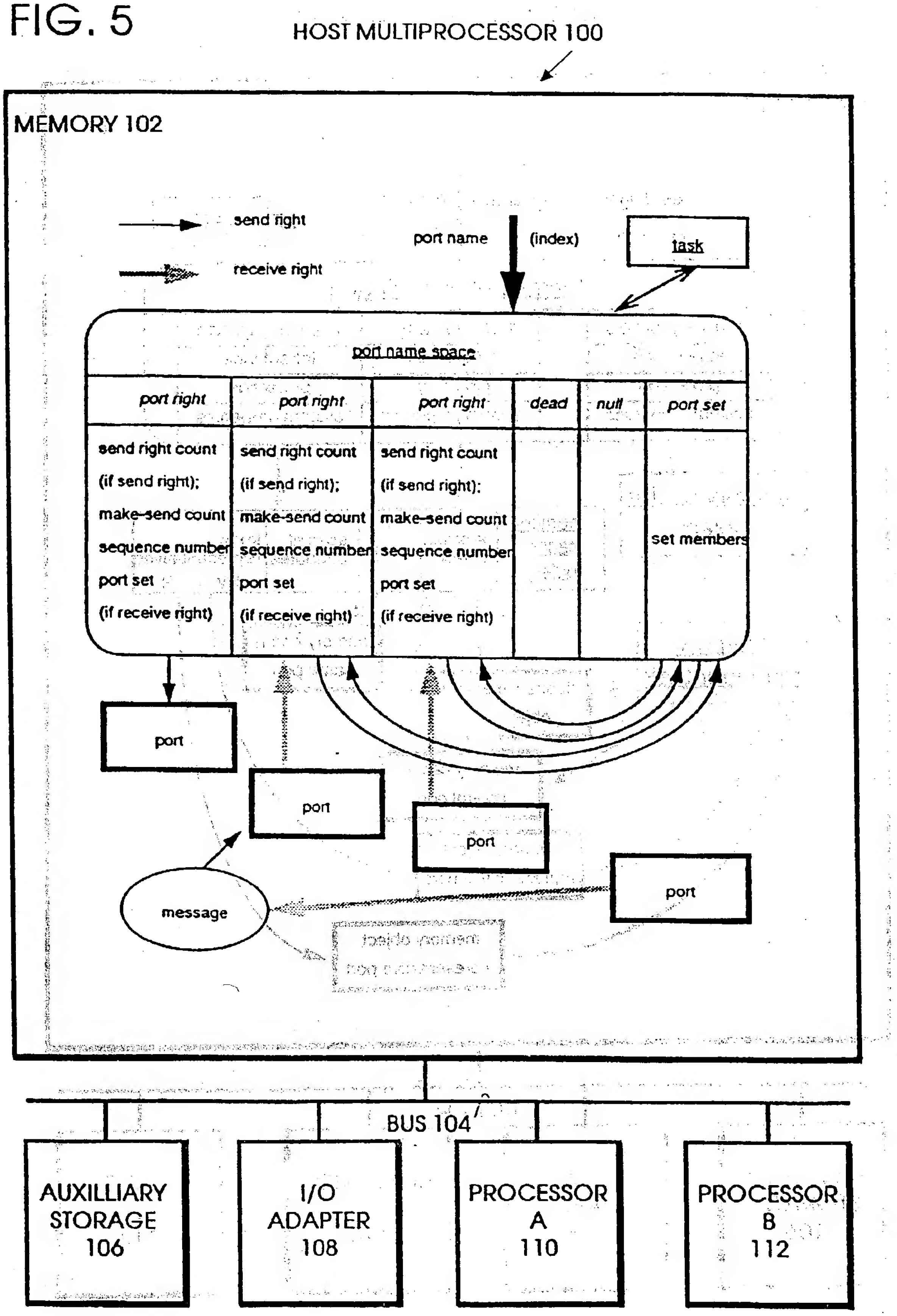

HOST MULTIPROCESSOR 100
MEMORY 102
send right
receive right
port name
(index)
task
port name space
port right
port right
port right
dead
null
port set
send right count
(if send right);
make-send count
sequence number
port set
(if receive right)
send right count
(if send right);
make-send count
sequence number
port set
(if receive right)
send right count
(if send right);
make-send count
sequence number
port set
(if receive right)
set members
port
port
port
port
message
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 6

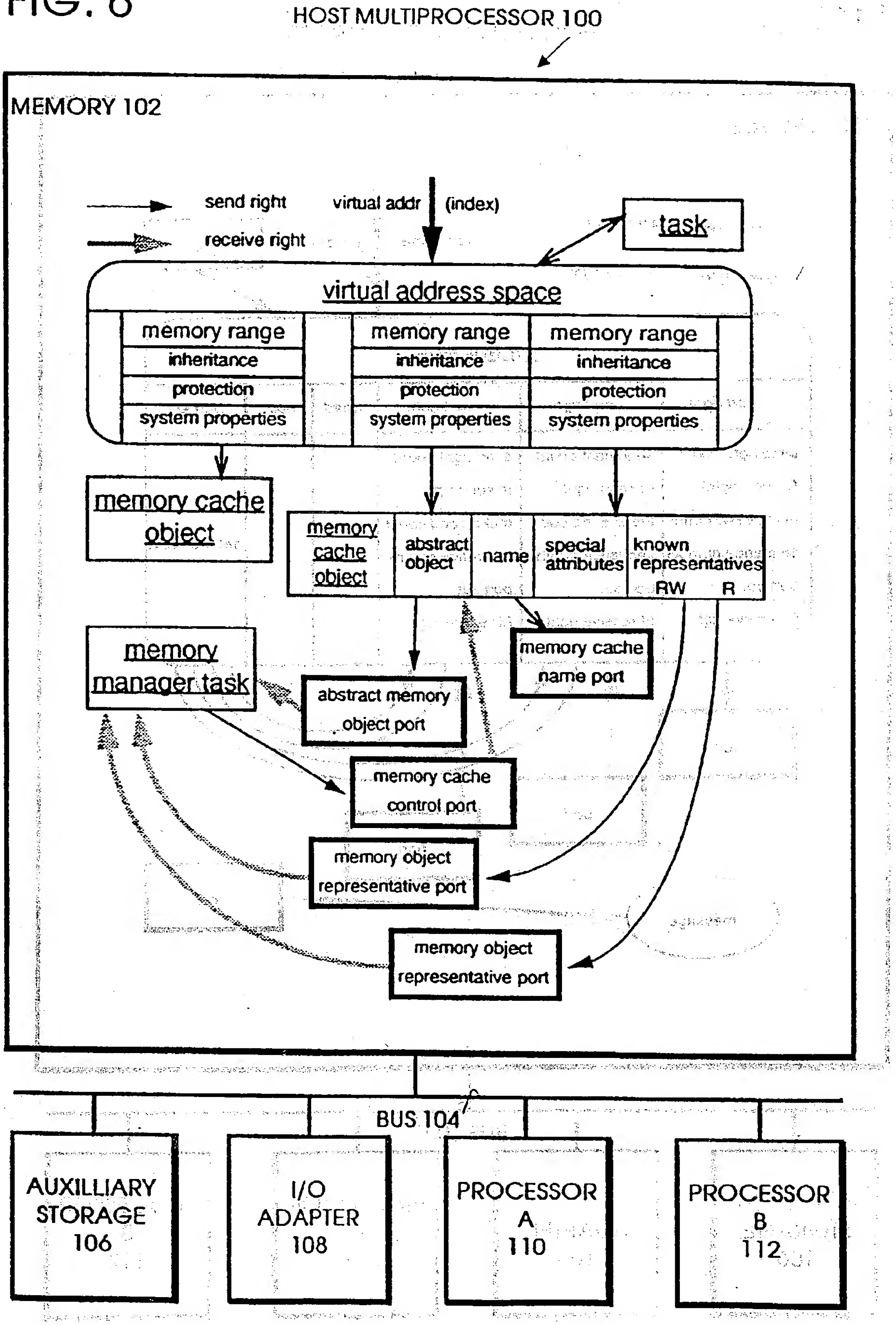
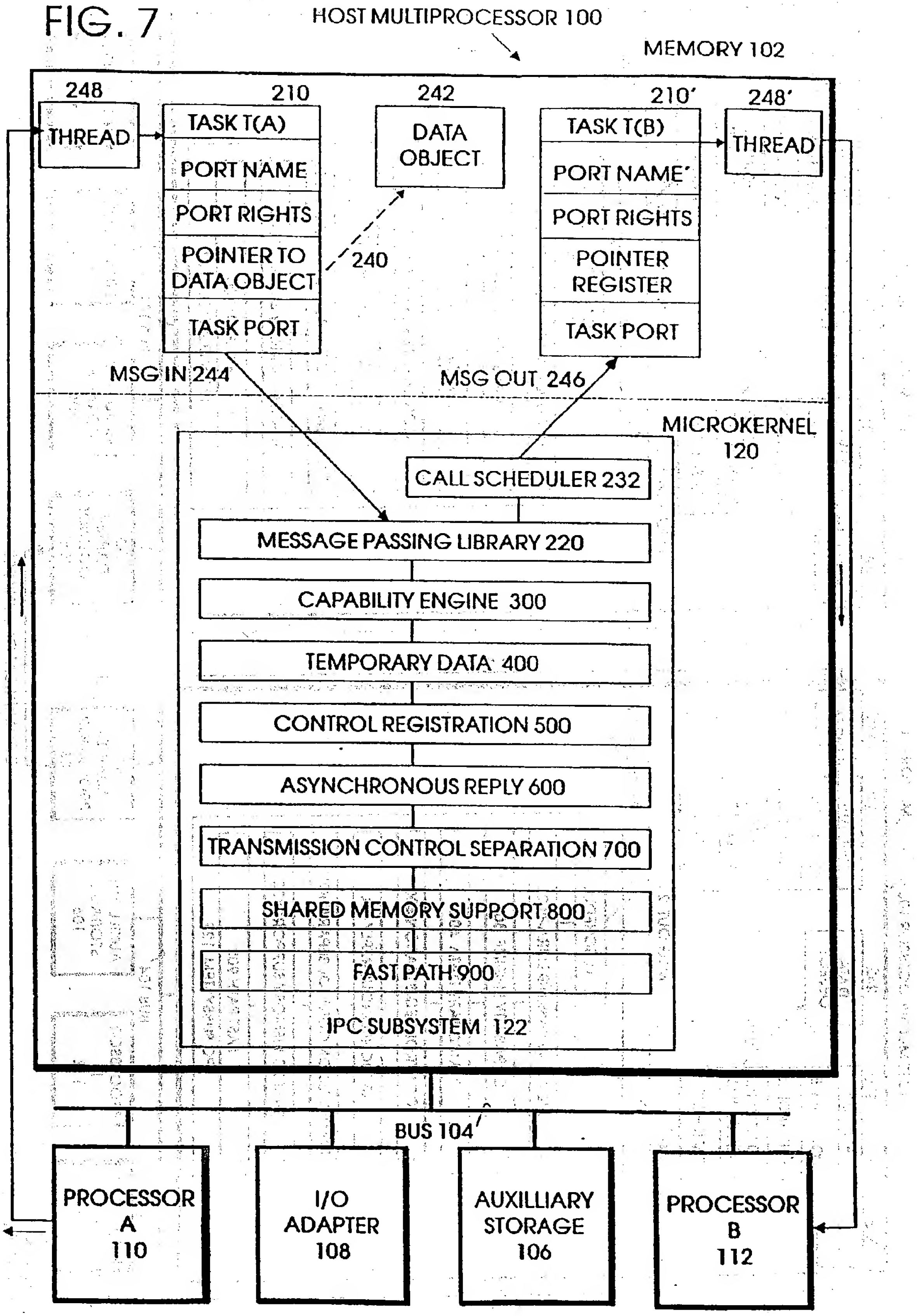

HOST MULTIPROCESSOR 100
MEMORY 102
send right
receive right
virtual addr
(index)
task
virtual address space
memory range
inheritance
protection
system properties
memory range
inheritance
protection
system properties
memory range
inheritance
protection
system properties
memory cache object
memory cache object
abstract object
name
special attributes
known representatives
RW
R
memory manager task
abstract memory object port
memory cache name port
memory cache control port
memory object representative port
memory object representative port
BUS 104
AUXILLIARY STORAGE 106
I/O ADAPTER 108
PROCESSOR A 110
PROCESSOR B 112

FIG. 7


HOST MULTIPROCESSOR 100
MEMORY 102
248
THREAD
210
TASK T(A)
PORT NAME
PORT RIGHTS
POINTER TO DATA OBJECT
TASK PORT
242
DATA OBJECT
240
210'
TASK T(B)
PORT NAME'
PORT RIGHTS
POINTER REGISTER
TASK PORT
248'
THREAD
MSG IN 244
MSG OUT 246
MICROKERNEL 120
CALL SCHEDULER 232
MESSAGE PASSING LIBRARY 220
CAPABILITY ENGINE 300
TEMPORARY DATA 400
CONTROL REGISTRATION 500
ASYNCHRONOUS REPLY 600
TRANSMISSION CONTROL SEPARATION 700
SHARED MEMORY SUPPORT 800
FAST PATH 900
IPC SUBSYSTEM 122
BUS 104
PROCESSOR A 110
I/O ADAPTER 108
AUXILLIARY STORAGE 106
PROCESSOR B 112

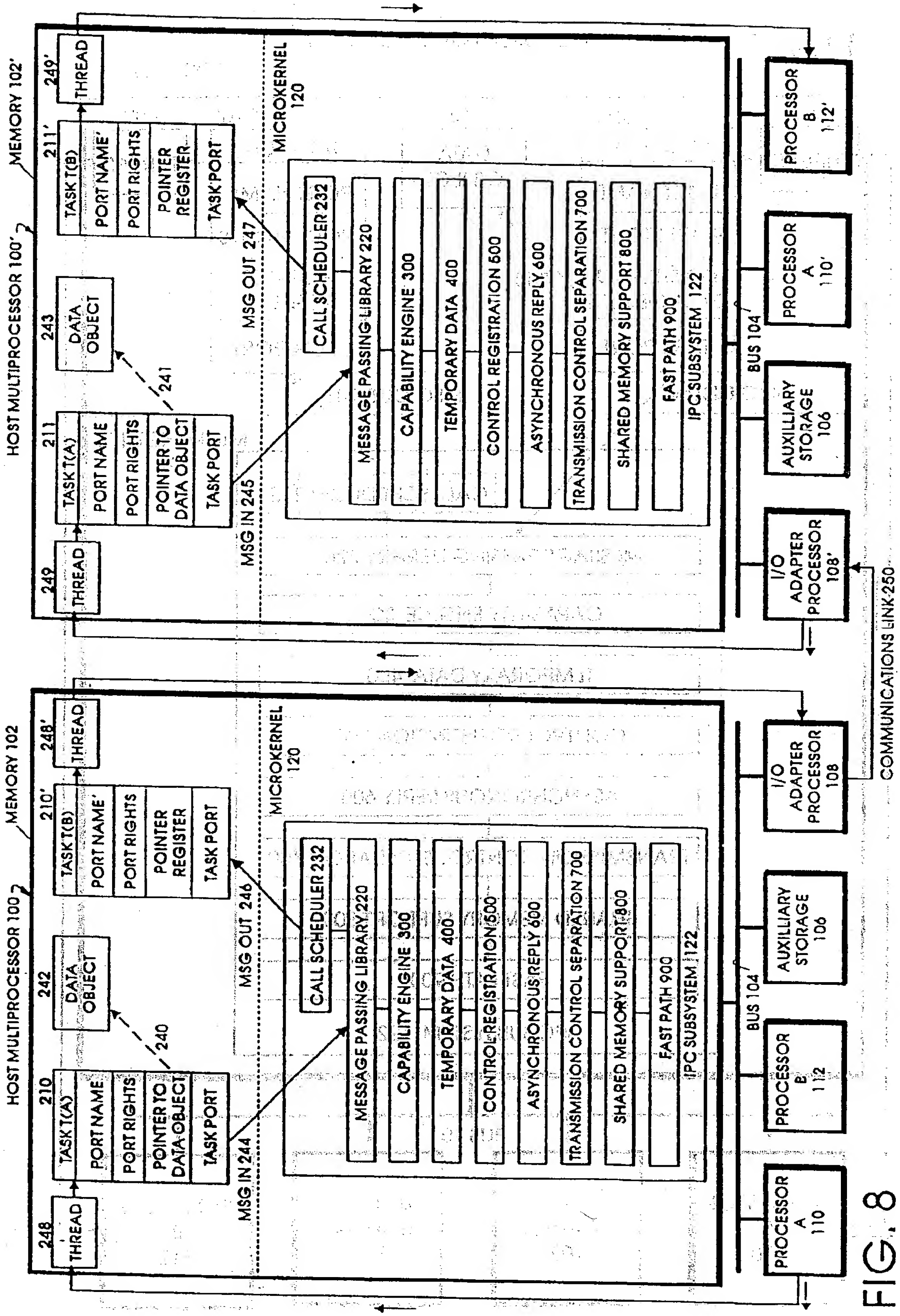

HOST MULTIPROCESSOR 100
MEMORY 102
248
THREAD
210
TASK T(A)
PORT NAME
PORT RIGHTS
POINTER TO DATA OBJECT
TASK PORT
240
242
DATA OBJECT
210'
TASK T(B)
PORT NAME'
PORT RIGHTS
POINTER REGISTER
TASK PORT
248'
THREAD
MSG IN 244
MSG OUT 246
MICROKERNEL 120
CALL SCHEDULER 232
MESSAGE PASSING LIBRARY 220
CAPABILITY ENGINE 300
TEMPORARY DATA 400
CONTROL REGISTRATION 500
ASYNCHRONOUS REPLY 600
TRANSMISSION CONTROL SEPARATION 700
SHARED MEMORY SUPPORT 800
FAST PATH 900
IPC SUBSYSTEM 122
BUS 104
PROCESSOR A 110
PROCESSOR B 112
AUXILLIARY STORAGE 106
I/O ADAPTER PROCESSOR 108
COMMUNICATIONS LINK 250
HOST MULTIPROCESSOR 100'
MEMORY 102'
249
THREAD
211
TASK T(A)
PORT NAME
PORT RIGHTS
POINTER TO DATA OBJECT
TASK PORT
241
243
DATA OBJECT
211'
TASK T(B)
PORT NAME'
PORT RIGHTS
POINTER REGISTER
TASK PORT
249'
THREAD
MSG IN 245
MSG OUT 247
MICROKERNEL 120
CALL SCHEDULER 232
MESSAGE PASSING LIBRARY 220
CAPABILITY ENGINE 300
TEMPORARY DATA 400
CONTROL REGISTRATION 500
ASYNCHRONOUS REPLY 600
TRANSMISSION CONTROL SEPARATION 700
SHARED MEMORY SUPPORT 800
FAST PATH 900
IPC SUBSYSTEM 122
BUS 104
I/O ADAPTER PROCESSOR 108'
AUXILLIARY STORAGE 106
PROCESSOR A 110'
PROCESSOR B 112'

FIG. 8

FIG. 9

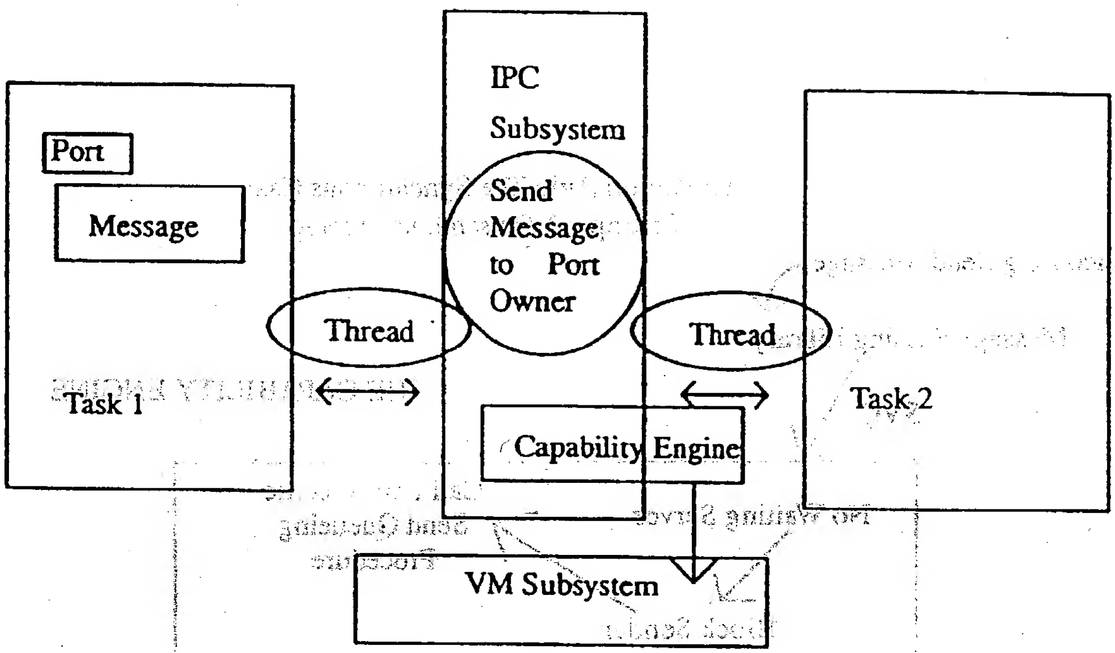

IPC
Subsystem
Port
Message
Send
Message
to Port
Owner
Thread
Thread
Task 1
Task 2
Capability Engine
VM Subsystem

FIG. 10

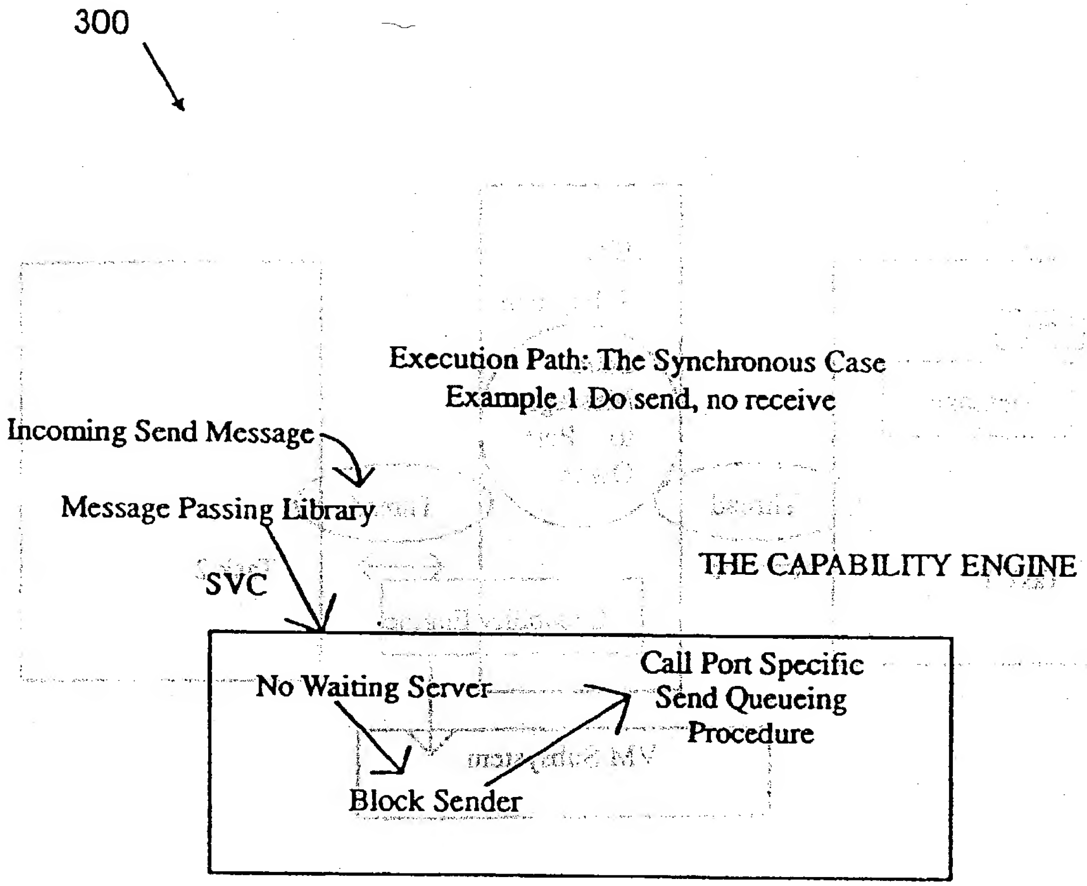

300
Execution Path: The Synchronous Case
Example 1 Do send, no receive
Incoming Send Message
Message Passing Library
SVC
THE CAPABILITY ENGINE
No Waiting Server
Call Port Specific
Send Queueing
Procedure
Block Sender

FIG. 11

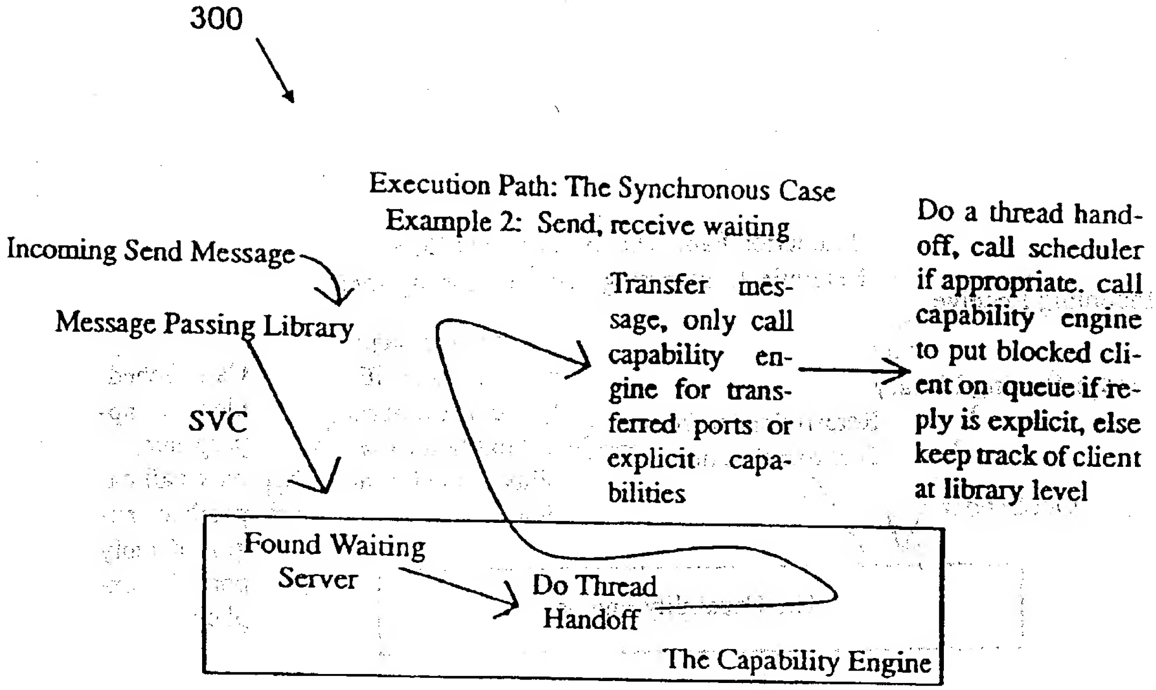

300
Execution Path: The Synchronous Case
Example 2: Send, receive waiting
Incoming Send Message
Message Passing Library
SVC
Transfer message, only call capability engine for transferred ports or explicit capabilities
Do a thread handoff, call scheduler if appropriate. call capability engine to put blocked client on queue if reply is explicit, else keep track of client at library level
Found Waiting Server
Do Thread Handoff
The Capability Engine

FIG. 12

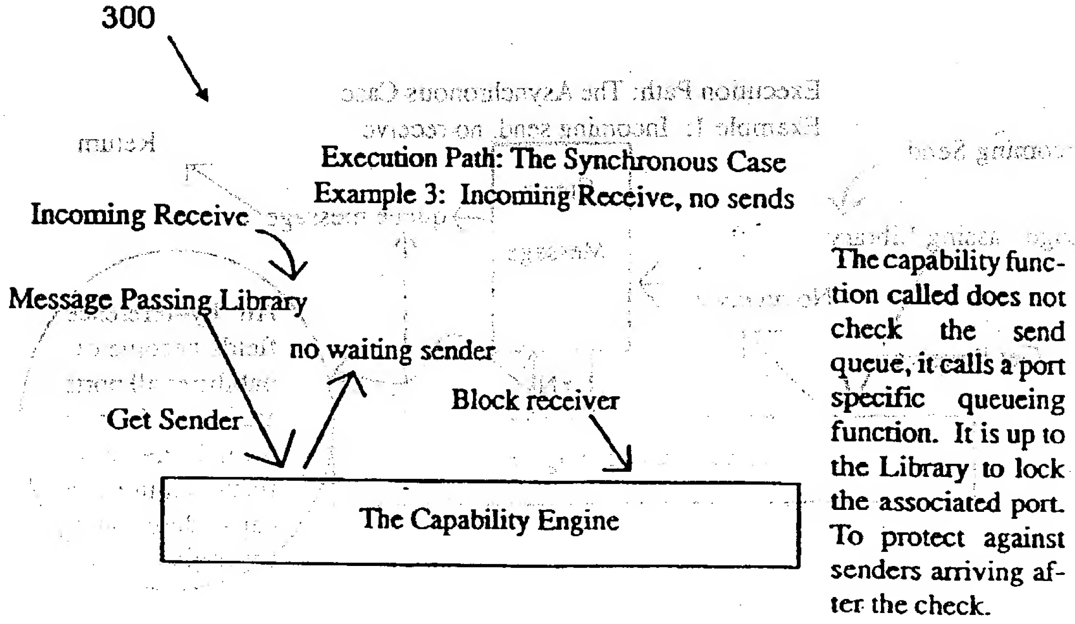

300
Execution Path: The Synchronous Case
Example 3: Incoming Receive, no sends
Incoming Receive
Message Passing Library
no waiting sender
Get Sender
Block receiver
The Capability Engine
The capability function called does not check the send queue, it calls a port specific queueing function. It is up to the Library to lock the associated port. To protect against senders arriving after the check.

FIG. 13

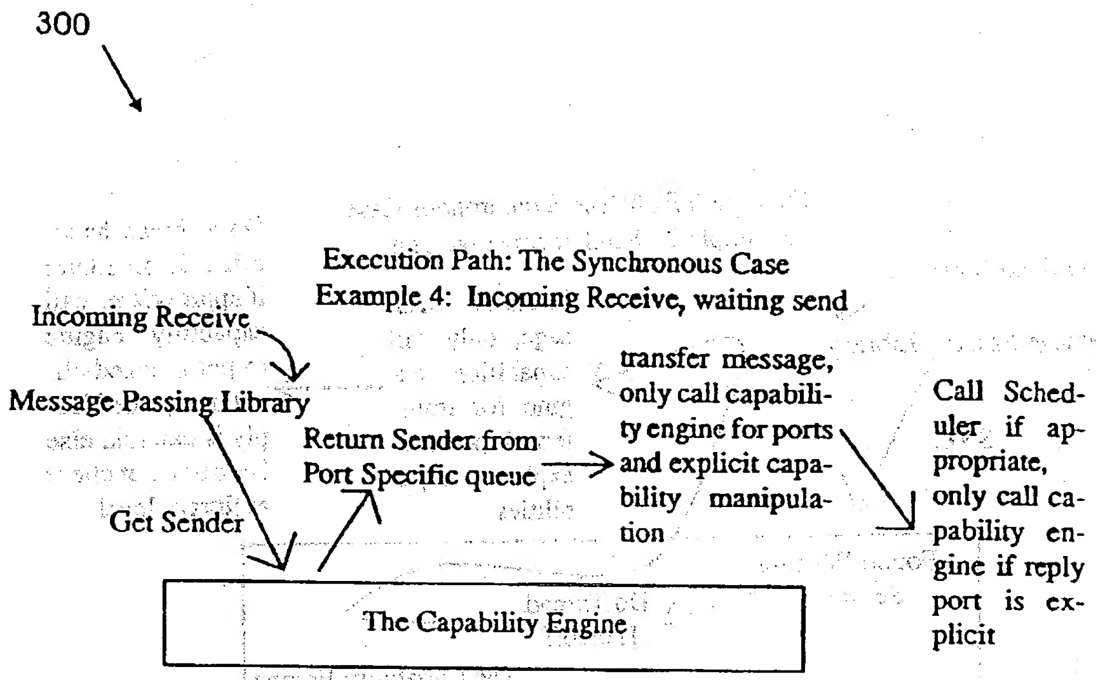

300
Execution Path: The Synchronous Case
Example 4: Incoming Receive, waiting send
Incoming Receive
Message Passing Library
Get Sender
Return Sender from Port Specific queue
transfer message, only call capability engine for ports and explicit capability manipulation
Call Scheduler if appropriate, only call capability engine if reply port is explicit
The Capability Engine

FIG. 14

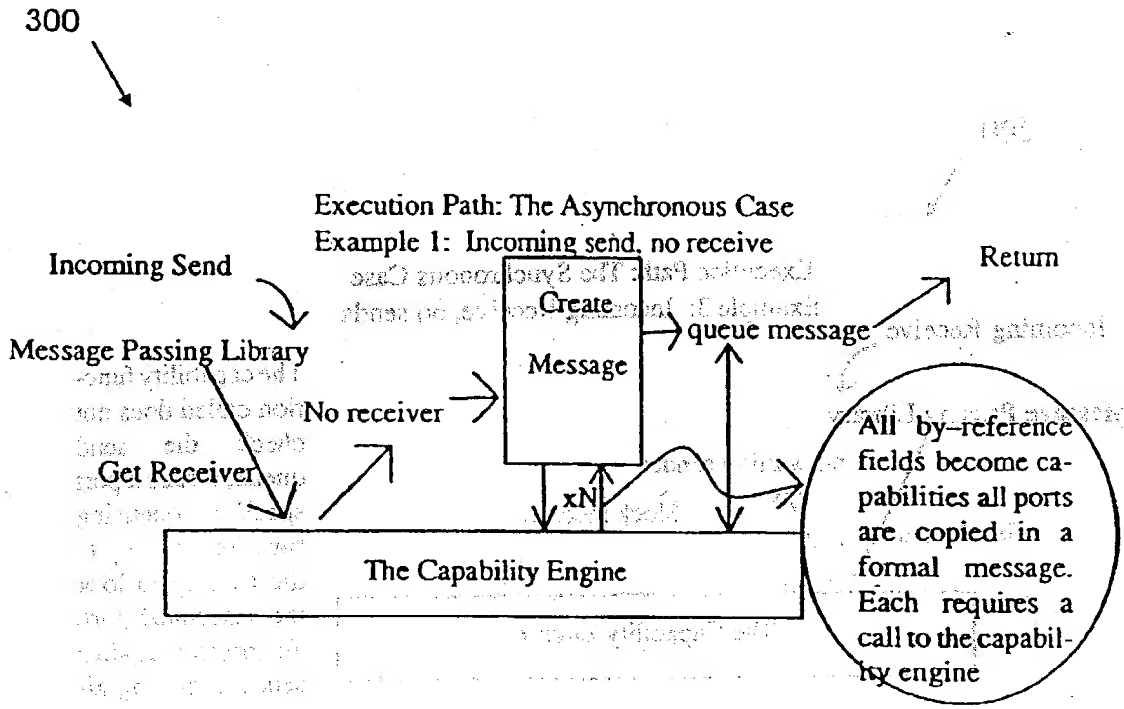

300
Execution Path: The Asynchronous Case
Example 1: Incoming send, no receive
Incoming Send
Message Passing Library
Get Receiver
No receiver
Create Message
queue message
Return
xN
All by-reference fields become capabilities all ports are copied in a formal message. Each requires a call to the capability engine
The Capability Engine

FIG. 15

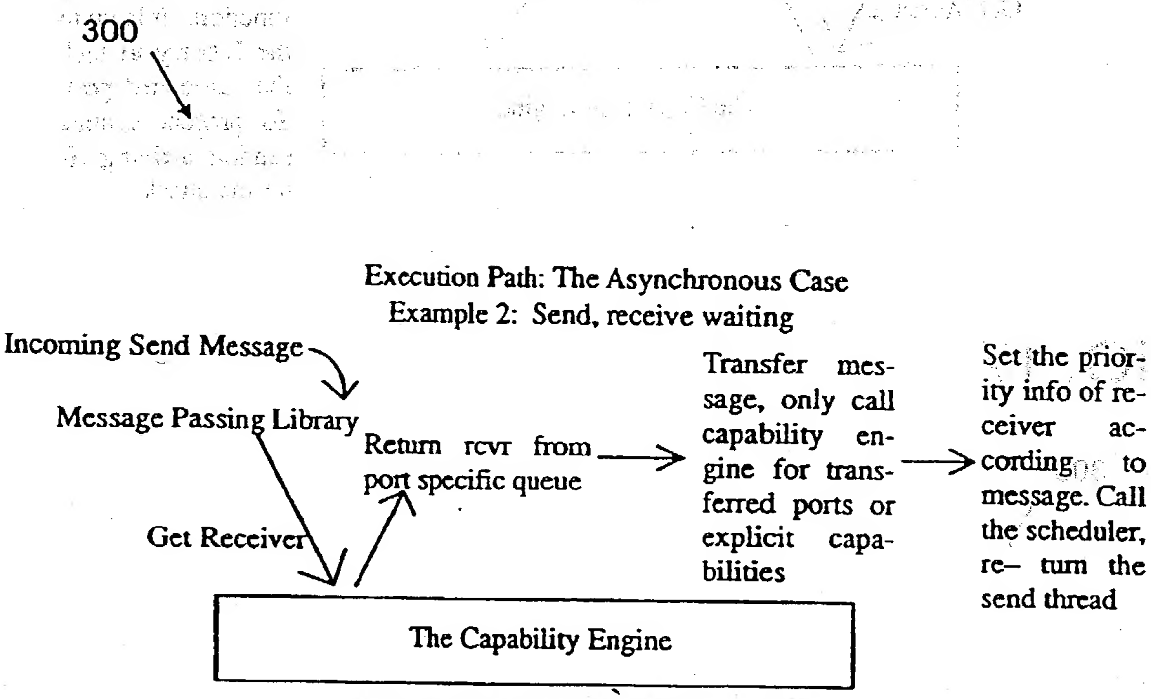

300
Execution Path: The Asynchronous Case
Example 2: Send, receive waiting
Incoming Send Message
Message Passing Library
Get Receiver
Return rcvr from port specific queue
Transfer message, only call capability engine for transferred ports or explicit capabilities
Set the priority info of receiver according to message. Call the scheduler, re– turn the send thread
The Capability Engine

FIG. 16

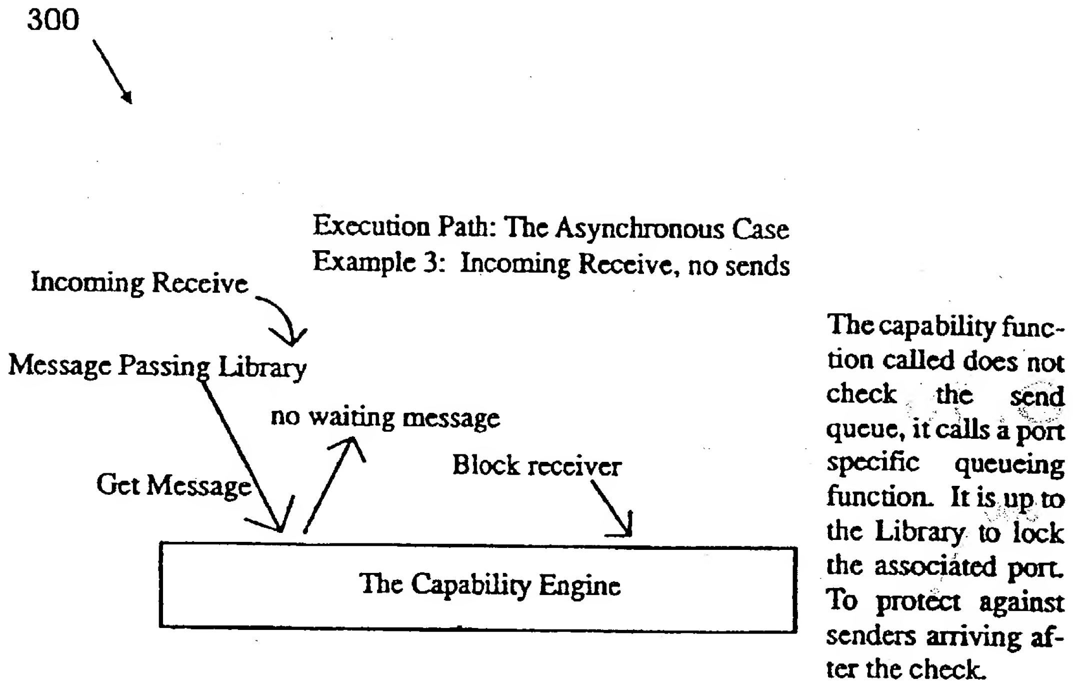

300
Execution Path: The Asynchronous Case
Example 3: Incoming Receive, no sends
Incoming Receive
Message Passing Library
no waiting message
Block receiver
Get Message
The Capability Engine
The capability function called does not check the send queue, it calls a port specific queueing function. It is up to the Library to lock the associated port. To protect against senders arriving after the check.

FIG. 17

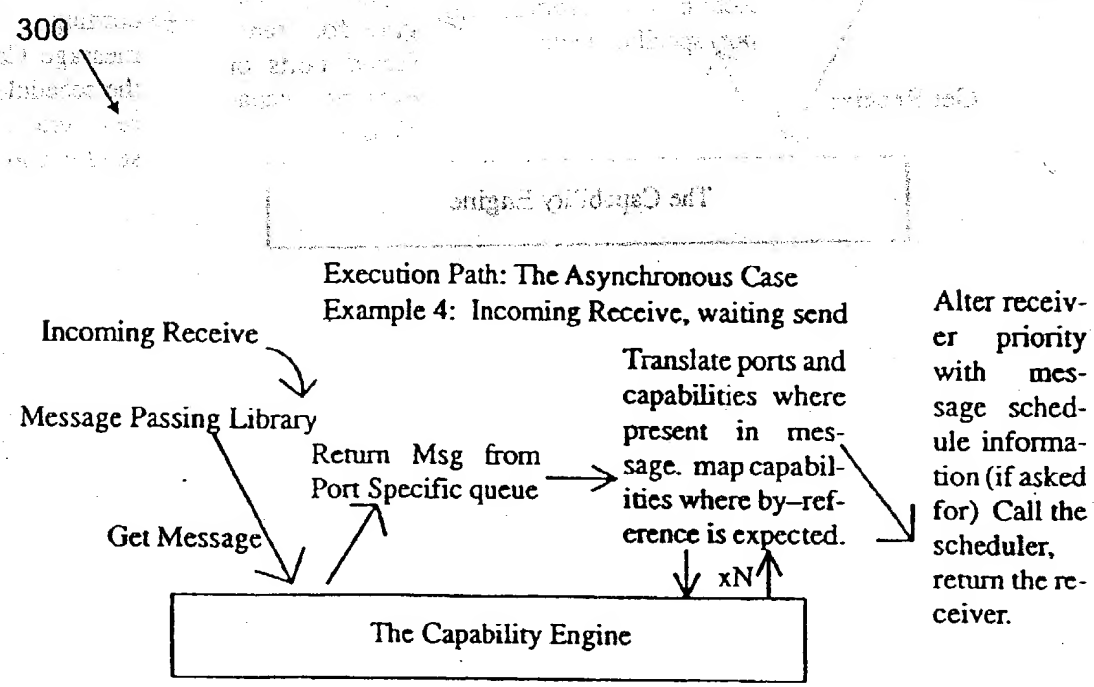

300
Execution Path: The Asynchronous Case
Example 4: Incoming Receive, waiting send
Incoming Receive
Message Passing Library
Return Msg from Port Specific queue
Translate ports and capabilities where present in message. map capabilities where by–reference is expected.
Alter receiver priority with message schedule information (if asked for) Call the scheduler, return the receiver.
Get Message
xN
The Capability Engine

FIG. 18

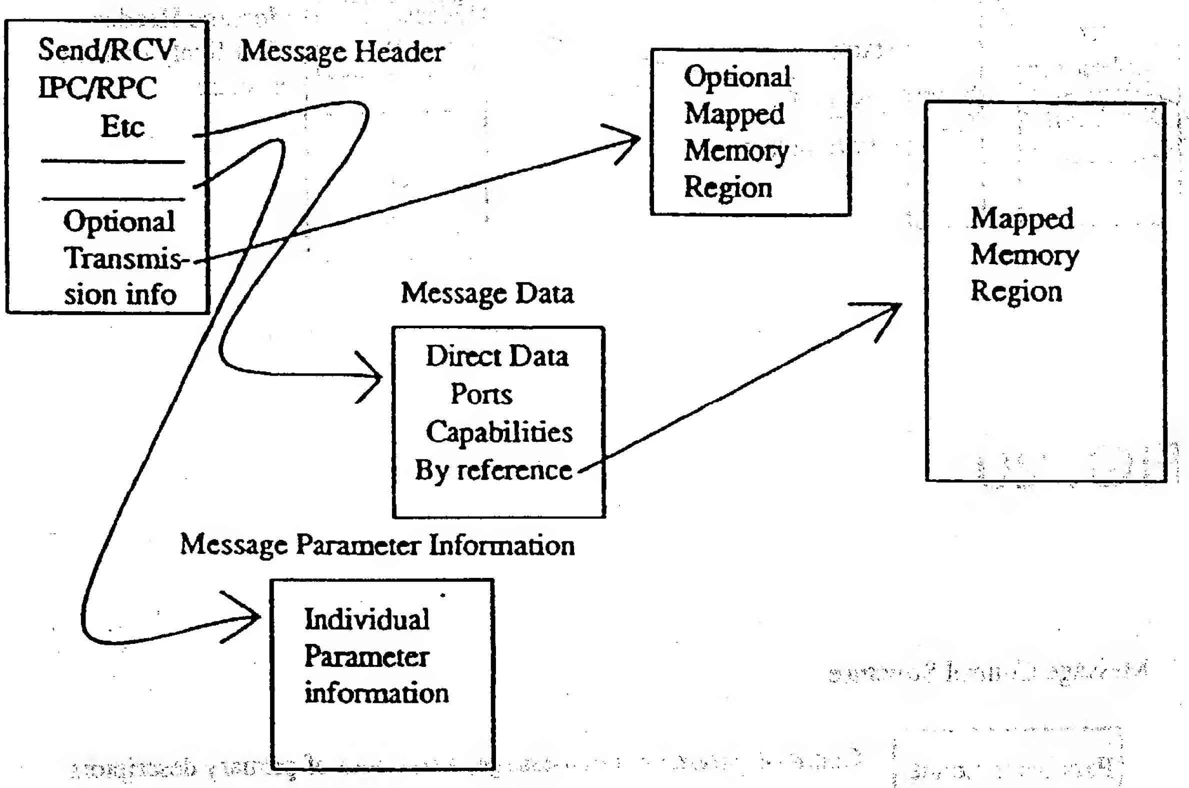

Send/RCV
IPC/RPC
Etc
Optional
Transmis-
sion info
Message Header
Optional
Mapped
Memory
Region
Mapped
Memory
Region
Message Data
Direct Data
Ports
Capabilities
By reference
Message Parameter Information
Individual
Parameter
information

FIG. 19

Typical Call of a By Proxy User Level Library

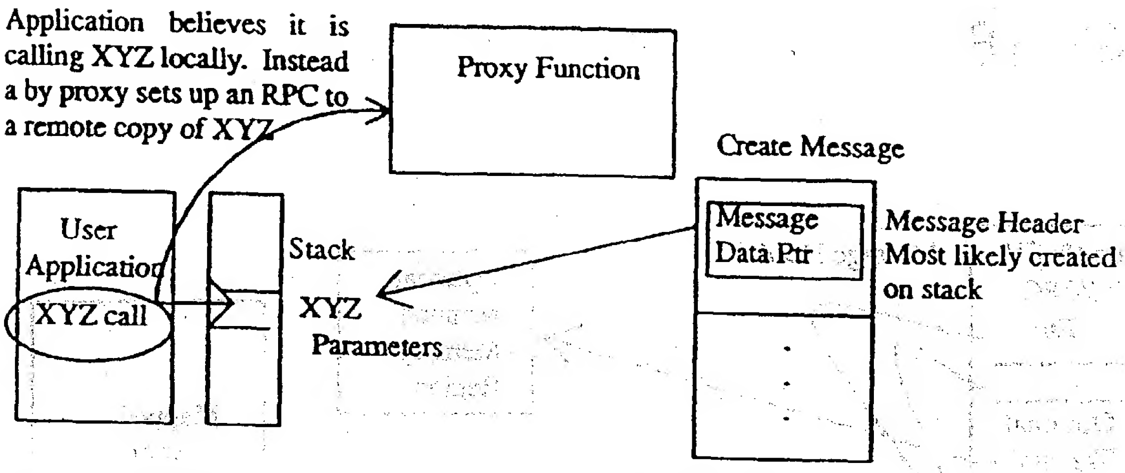

Application believes it is calling XYZ locally. Instead a by proxy sets up an RPC to a remote copy of XYZ
Proxy Function
Create Message
User Application
XYZ call
Stack
XYZ Parameters
Message Data Ptr
Message Header Most likely created on stack

FIG. 20

Message Control Structure

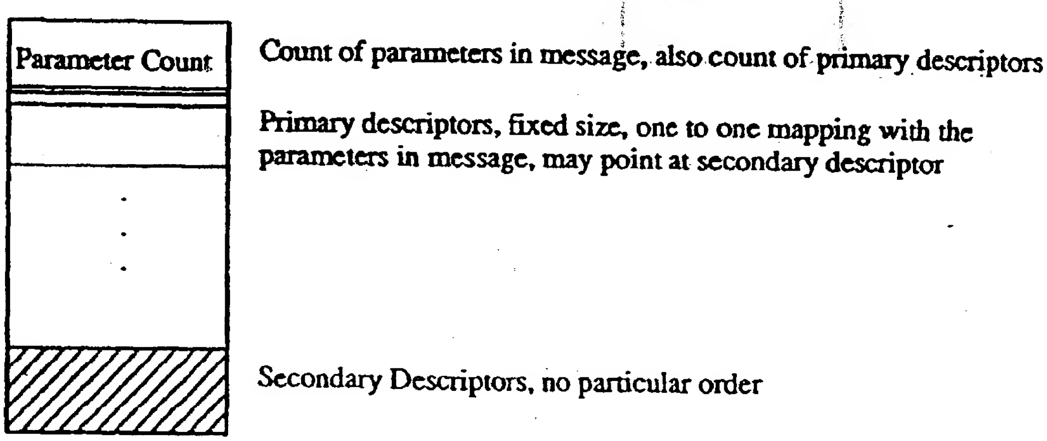

Parameter Count
Count of parameters in message, also count of primary descriptors
Primary descriptors, fixed size, one to one mapping with the parameters in message, may point at secondary descriptor
Secondary Descriptors, no particular order

FIG. 21

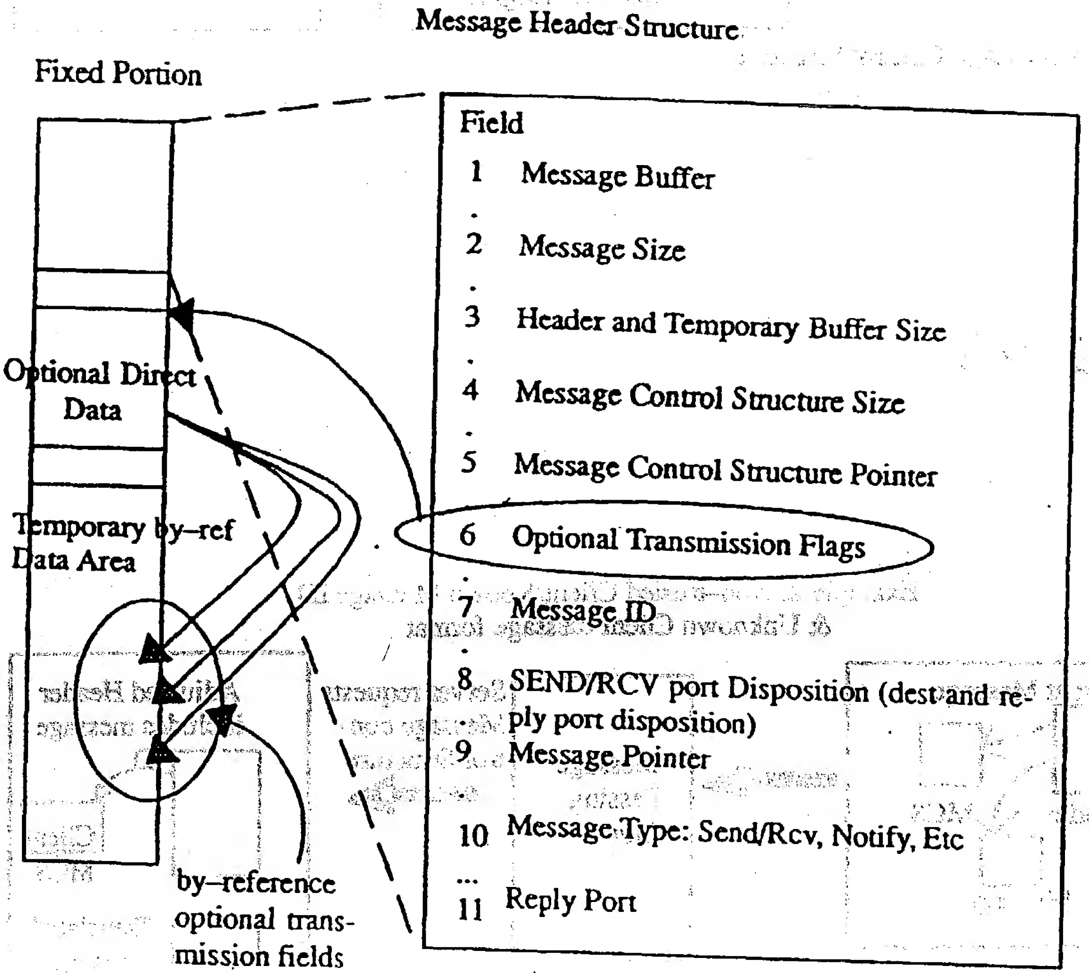
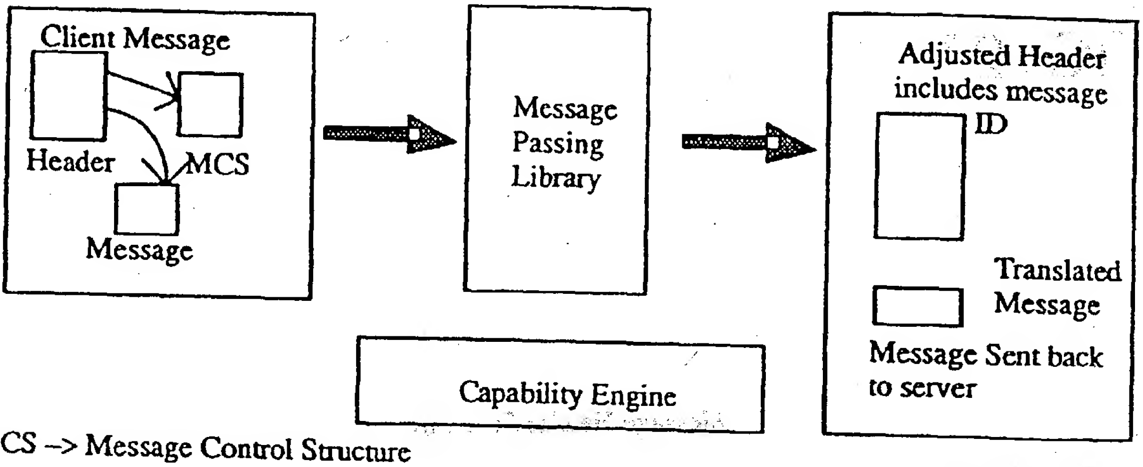

Message Header Structure
Fixed Portion
Optional Direct Data
Temporary by–ref Data Area
by–reference optional transmission fields
Field
1 Message Buffer
2 Message Size
3 Header and Temporary Buffer Size
4 Message Control Structure Size
5 Message Control Structure Pointer
6 Optional Transmission Flags
7 Message ID
8 SEND/RCV port Disposition (dest and reply port disposition)
9 Message Pointer
10 Message Type: Send/Rcv, Notify, Etc
11 Reply Port

FIG. 22

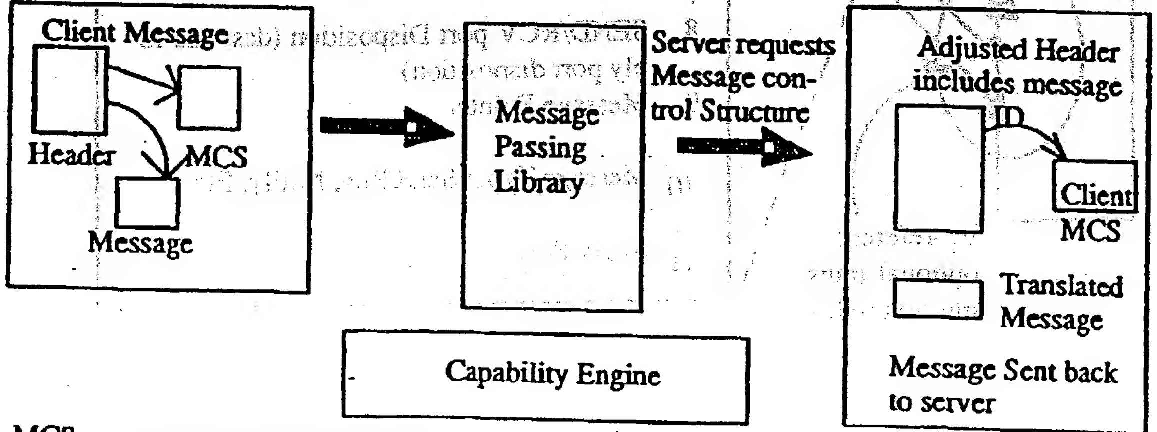
Example 1: Trusted Client/Known Message ID



MCS –> Message Control Structure

FIG. 23

Example 2: Non–trusted Client/Known Message ID & Unknown Client message format


Client Message
Header
MCS
Message
Message Passing Library
Server requests Message control Structure
Capability Engine
Adjusted Header includes message ID
Client MCS
Translated Message
Message Sent back to server

MCS –> message control structure

FIG. 24

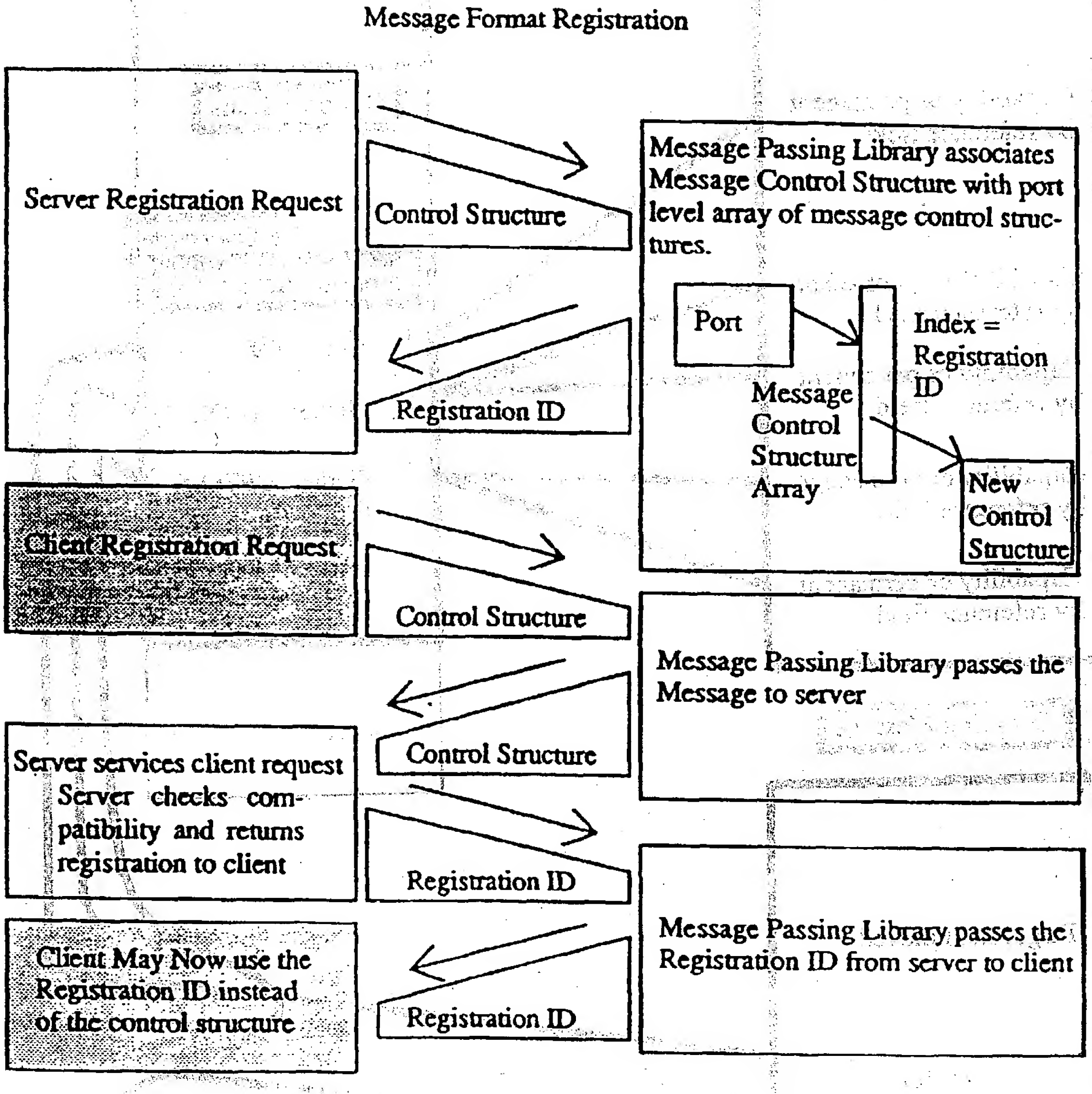

Message Format Registration
Server Registration Request
Control Structure
Message Passing Library associates Message Control Structure with port level array of message control structures.
Port
Index = Registration ID
Message Control Structure Array
New Control Structure
Registration ID
Client Registration Request
Control Structure
Message Passing Library passes the Message to server
Control Structure
Server services client request Server checks compatibility and returns registration to client
Registration ID
Message Passing Library passes the Registration ID from server to client
Registration ID
Client May Now use the Registration ID instead of the control structure

FIG. 25

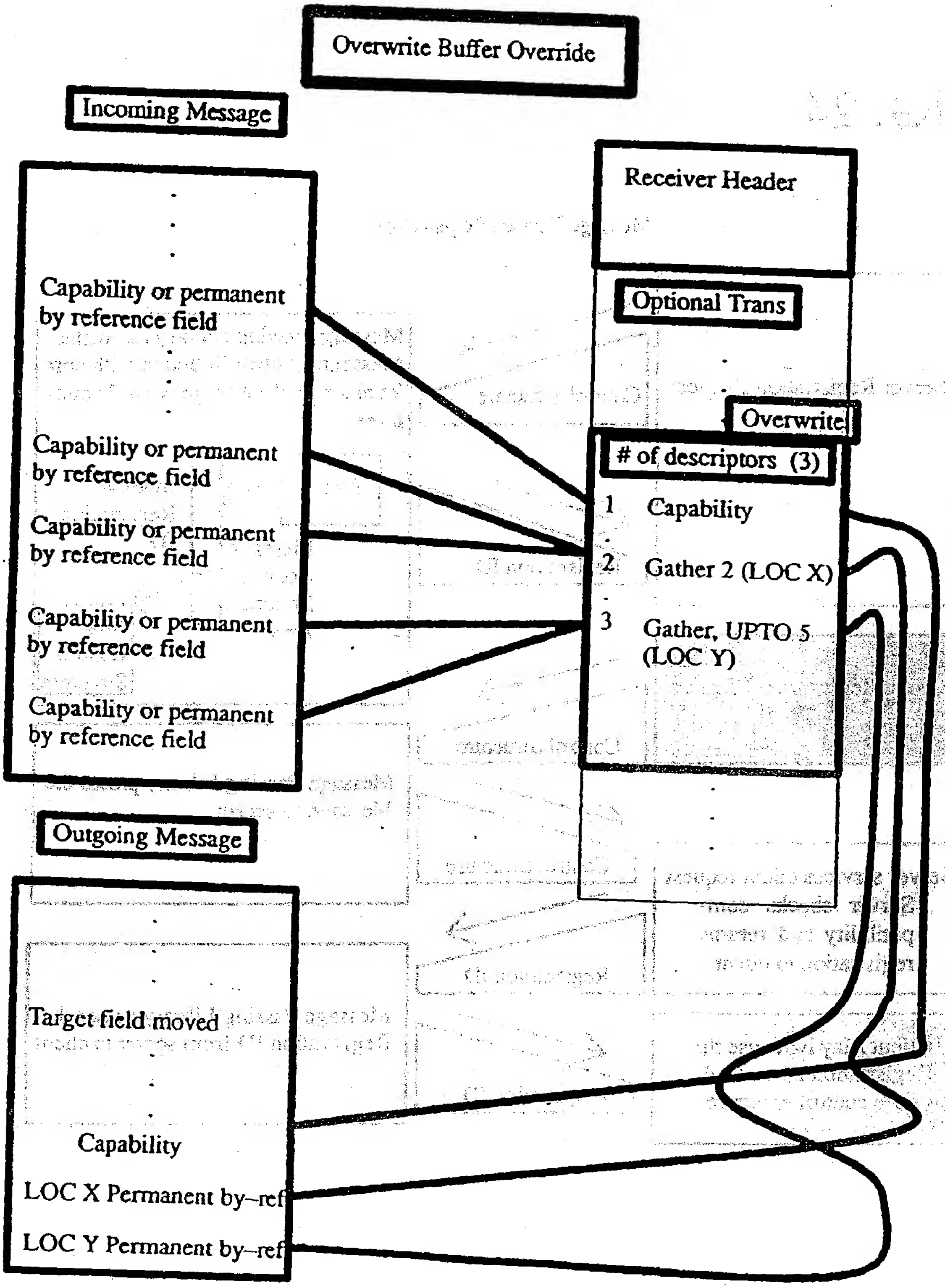

Overwrite Buffer Override
Incoming Message
Capability or permanent by reference field
Capability or permanent by reference field
Capability or permanent by reference field
Capability or permanent by reference field
Capability or permanent by reference field
Receiver Header
Optional Trans
Overwrite
# of descriptors (3)
1 Capability
2 Gather 2 (LOC X)
3 Gather, UPTO 5 (LOC Y)
Outgoing Message
Target field moved
Capability
LOC X Permanent by–ref
LOC Y Permanent by–ref

FIG. 26

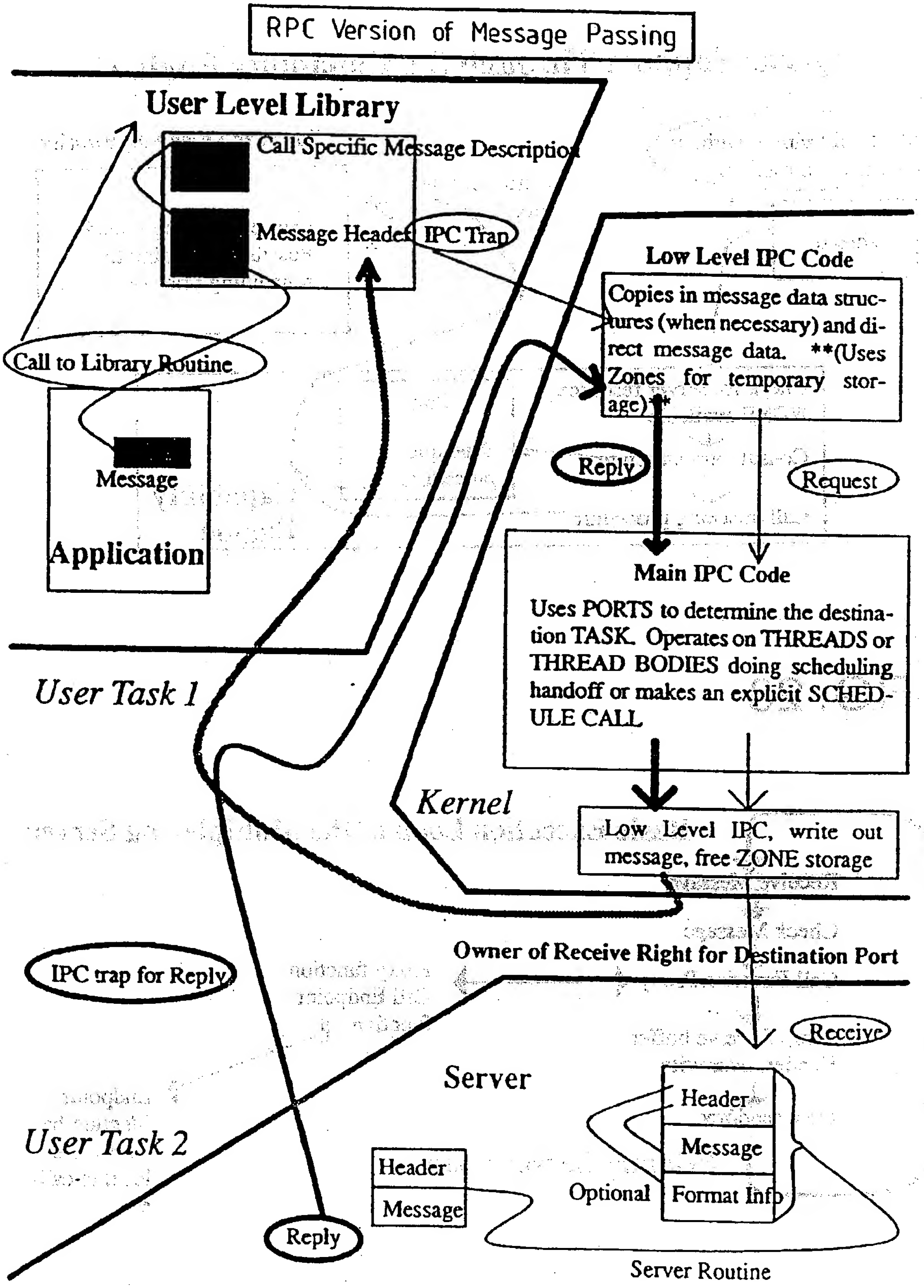

RPC Version of Message Passing
User Level Library
Call Specific Message Description
Message Header
IPC Trap
Call to Library Routine
Message
Application
User Task 1
Low Level IPC Code
Copies in message data structures (when necessary) and direct message data. **(Uses Zones for temporary storage)**
Reply
Request
Main IPC Code
Uses PORTS to determine the destination TASK. Operates on THREADS or THREAD BODIES doing scheduling handoff or makes an explicit SCHEDULE CALL
Kernel
Low Level IPC, write out message, free ZONE storage
Owner of Receive Right for Destination Port
IPC trap for Reply
Receive
Server
Header
Message
Optional
Format Info
User Task 2
Header
Message
Reply
Server Routine

FIG. 27

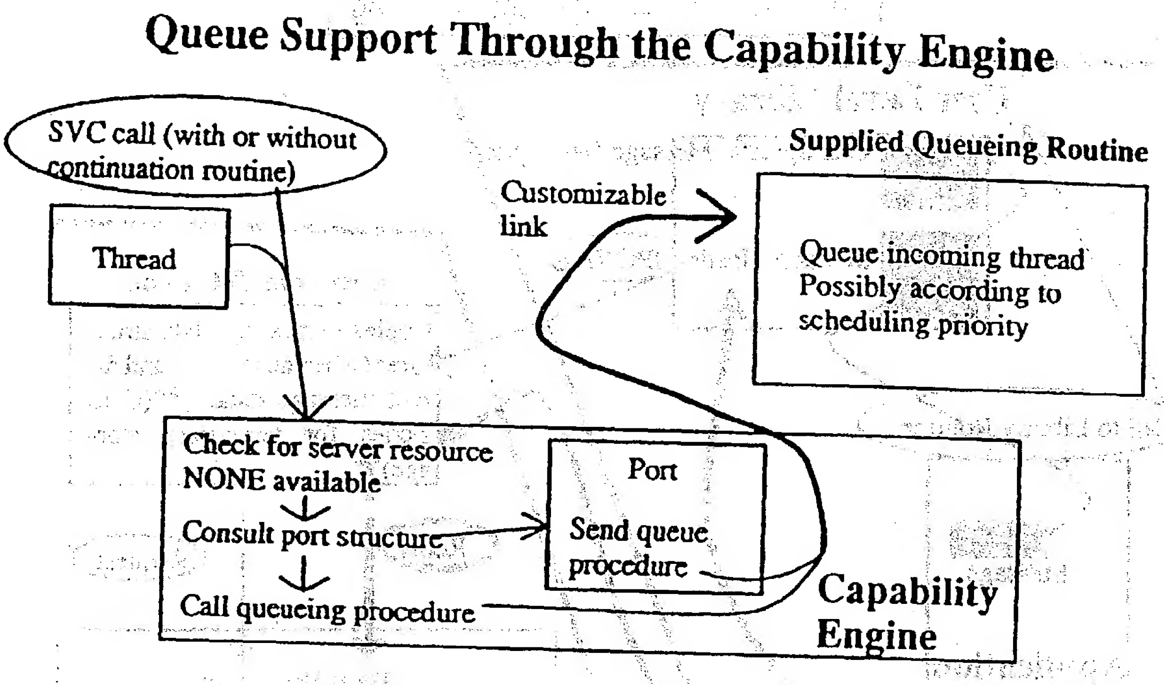

Queue Support Through the Capability Engine
SVC call (with or without continuation routine)
Thread
Customizable link
Supplied Queueing Routine
Queue incoming thread
Possibly according to
scheduling priority
Check for server resource
NONE available
Consult port structure
Call queueing procedure
Port
Send queue procedure
Capability Engine

FIG. 28

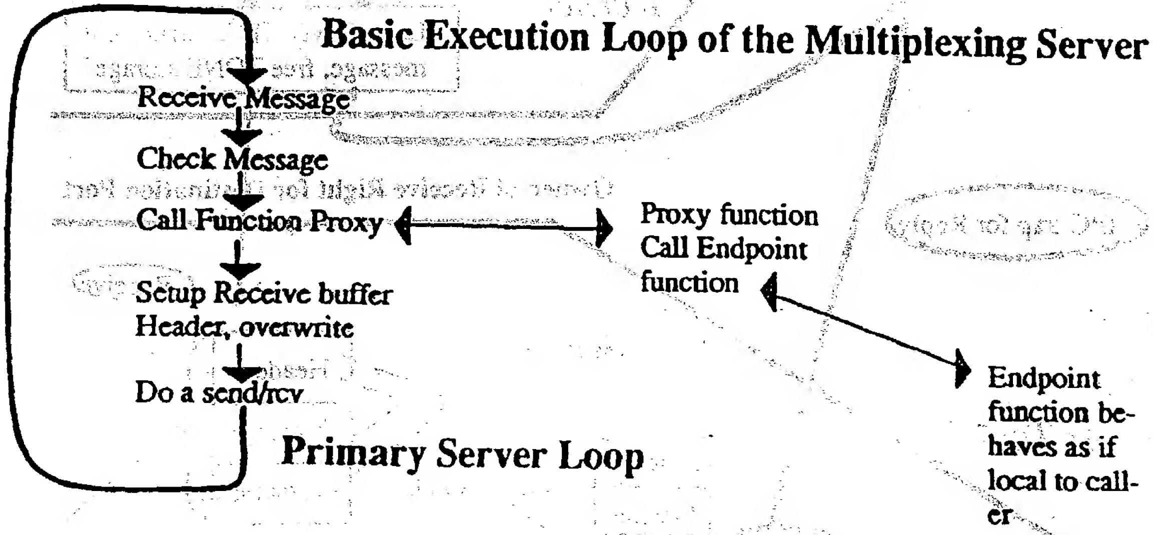

Basic Execution Loop of the Multiplexing Server
Receive Message
Check Message
Call Function Proxy
Setup Receive buffer
Header, overwrite
Do a send/rcv
Primary Server Loop
Proxy function
Call Endpoint
function
Endpoint
function be-
haves as if
local to call-
er

FIG. 29

Message Passing Library Anonymous Reply Algorithm

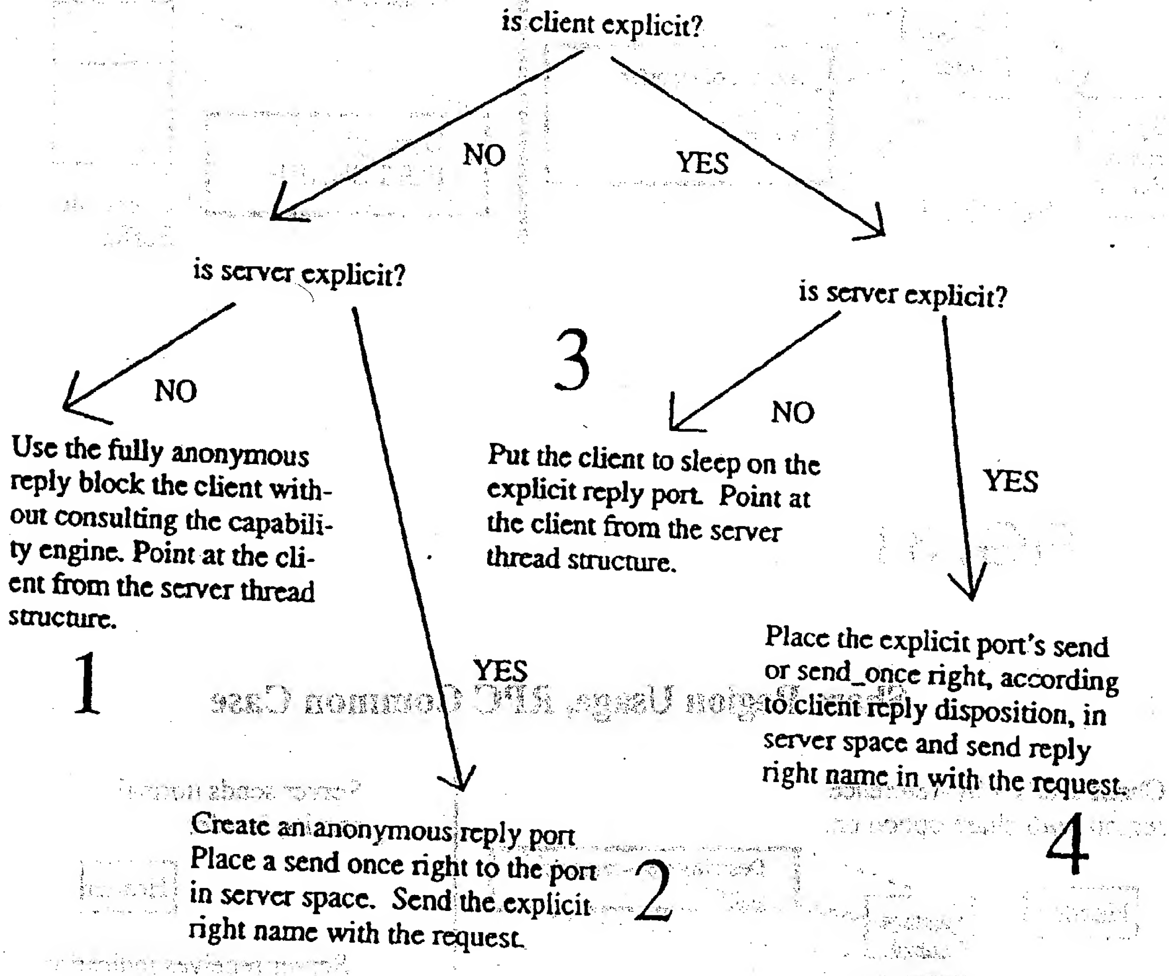

is client explicit?
NO
YES
is server explicit?
is server explicit?
3
NO
NO
Use the fully anonymous reply block the client without consulting the capability engine. Point at the client from the server thread structure.
Put the client to sleep on the explicit reply port. Point at the client from the server thread structure.
YES
1
YES
Place the explicit port's send or send_once right, according to client reply disposition, in server space and send reply right name in with the request.
Create an anonymous reply port Place a send once right to the port in server space. Send the explicit right name with the request.
2
4

FIG. 30

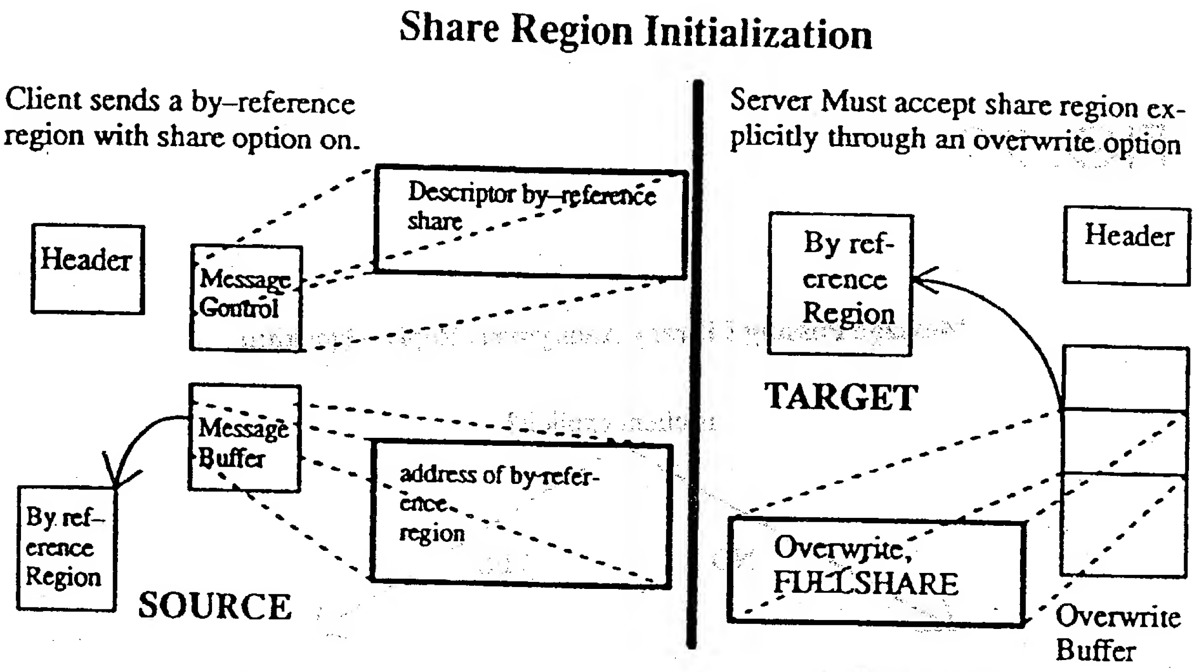
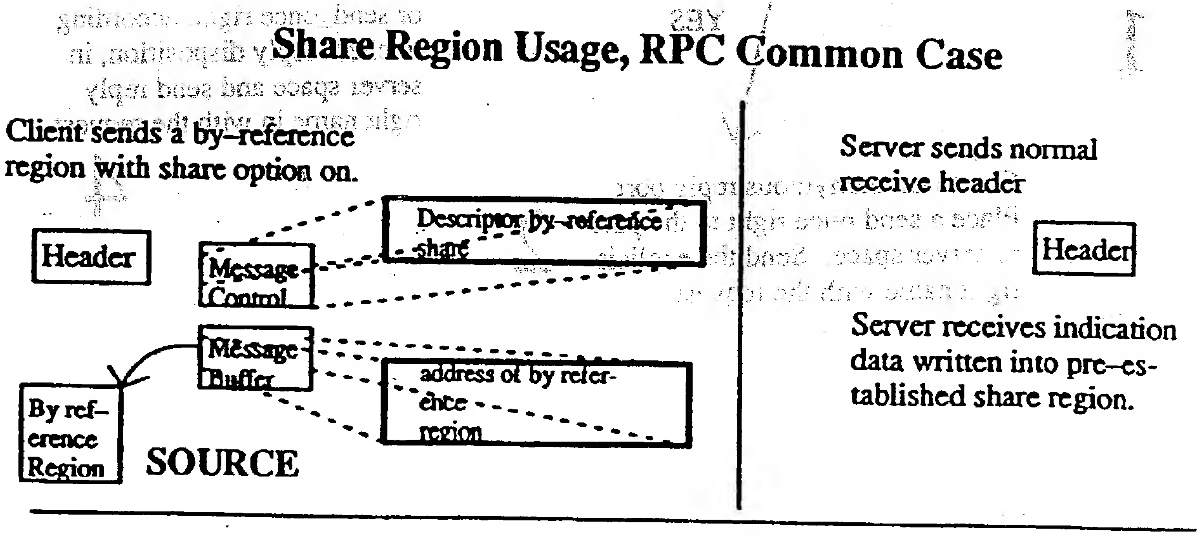

Share Region Initialization
Client sends a by–reference region with share option on.
Header
Message Control
Descriptor by–reference share
Message Buffer
address of by–reference region
By reference Region
SOURCE
Server Must accept share region explicitly through an overwrite option
By reference Region
Header
TARGET
Overwrite, FULLSHARE
Overwrite Buffer

FIG. 31


Share Region Usage, RPC Common Case
Client sends a by–reference region with share option on.
Header
Message Control
Descriptor by–reference share
Message Buffer
address of by reference region
By reference Region
SOURCE
Server sends normal receive header
Header
Server receives indication data written into pre–established share region.